Introduction to JAVA™

PROGRAMMING AND DATA STRUCTURES

COMPREHENSIVE VERSION

Y. DANIEL LIANG

Pearson

12th Edition

INTRODUCTION TO JAVA™ PROGRAMMING AND DATA STRUCTURES

COMPREHENSIVE VERSION

Twelfth Edition

Y. Daniel Liang

Georgia Southern University

Pearson

To Samantha, Michael, and Michelle

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided “as is” without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

Copyright © 2020, 2018, 2015 by Pearson Education, Inc. or its affiliates, 221 River Street, Hoboken, NJ 07030. All Rights Reserved. Manufactured in the United States of America. This publication is protected by copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights and Permissions department, please visit www.pearsoned.com/permissions/.

Acknowledgments of third-party content appear on the appropriate page within the text

PEARSON, ALWAYS LEARNING, and MYLAB are exclusive trademarks owned by Pearson Education, Inc. or its affiliates in the U.S. and/or other countries.

Unless otherwise indicated herein, any third-party trademarks, logos, or icons that may appear in this work are the property of their respective owners, and any references to third-party trademarks, logos, icons, or other trade dress are for demonstrative or descriptive purposes only. Such references are not intended to imply any sponsorship, endorsement, authorization, or promotion of Pearson’s products by the owners of such marks, or any relationship between the owner and Pearson Education, Inc., or its affiliates, authors, licensees, or distributors.

Library of Congress Cataloging-in-Publication Data

Names: Liang, Y. Daniel, author.
Title: Java programming and data structures / Y. Daniel Liang, Georgia Southern University.
Other titles: Introduction to Java programming and data structures
Description: Twelfth edition. Comprehensive version | Hoboken, NJ : Pearson, 2019. | Revised edition of: Introduction to Java programming and data structures / Y. Daniel Liang, Georgia Southern University. Eleventh edition. Comprehensive version. 2018. | Includes bibliographical references and index.
Identifiers: LCCN 2019038073 | ISBN 9780136520238 (paperback)
Subjects: LCSH: Java (Computer program language)
Classification: LCC QA76.73.J38 L52 2019 | DDC 005.13/3–dc23
LC record available at https://lccn.loc.gov/2019038073

ScoutAutomatedPrintCode

LLE ISBN
ISBN-10: 0-13-651996-2
ISBN-13: 978-0-13-651996-6

SE
ISBN-10: 0-13-652023-5
ISBN-13: 978-0-13-652023-8

Pearson

PREFACE

Dear Reader,

Many of you have provided feedback on earlier editions of this book, and your comments and suggestions have greatly improved the book. This edition has been substantially enhanced in presentation, organization, examples, exercises, and supplements.

fundamentals-first

The book is fundamentals first by introducing basic programming concepts and techniques before designing custom classes. The fundamental concepts and techniques of selection statements, loops, methods, and arrays are the foundation for programming. Building this strong foundation prepares students to learn object-oriented programming and advanced Java programming.

problem-driven

This book teaches programming in a problem-driven way that focuses on problem solving rather than syntax. We make introductory programming interesting by using thought-provoking problems in a broad context. The central thread of early chapters is on problem solving. Appropriate syntax and library are introduced to enable readers to write programs for solving the problems. To support the teaching of programming in a problem-driven way, the book provides a wide variety of problems at various levels of difficulty to motivate students. To appeal to students in all majors, the problems cover many application areas, including math, science, business, financial, gaming, animation, and multimedia.

data structures

The book seamlessly integrates programming, data structures, and algorithms into one text. It employs a practical approach to teach data structures. We first introduce how to use various data structures to develop efficient algorithms, and then show how to implement these data structures. Through implementation, students gain a deep understanding on the efficiency of data structures and on how and when to use certain data structures. Finally, we design and implement custom data structures for trees and graphs.

comprehensive version

brief version

AP Computer Science

The book is widely used in the introductory programming, data structures, and algorithms courses in the universities around the world. This *comprehensive version* covers fundamentals of programming, object-oriented programming, GUI programming, data structures, algorithms, concurrency, networking, database, and Web programming. It is designed to prepare students to become proficient Java programmers. A *brief version* (*Introduction to Java Programming*, Brief Version, Twelfth Edition) is available for a first course on programming, commonly known as CS1. The brief version contains the first 18 chapters of the comprehensive version. An AP version of the book is also available for high school students taking an AP Computer Science course.

examples and exercises

The best way to teach programming is *by example*, and the only way to learn programming is *by doing*. Basic concepts are explained by example and a large number of exercises with various levels of difficulty are provided for students to practice. For our programming courses, we assign programming exercises after each lecture.

Our goal is to produce a text that teaches problem solving and programming in a broad context using a wide variety of interesting examples. If you have any comments on and suggestions for improving the book, please email me.

Sincerely,

Y. Daniel Liang
y.daniel.liang@gmail.com
www.pearsonhighered.com/liang

ACM/IEEE Curricular 2013 and ABET Course Assessment

The new ACM/IEEE Computer Science Curricular 2013 defines the Body of Knowledge organized into 18 Knowledge Areas. To help instructors design the courses based on this book, we provide sample syllabi to identify the Knowledge Areas and Knowledge Units. The sample syllabi are for a three semester course sequence and serve as an example for institutional customization. The sample syllabi are accessible from the Instructor Resource Website.

Many of our users are from the ABET-accredited programs. A key component of the ABET accreditation is to identify the weakness through continuous course assessment against the course outcomes. We provide sample course outcomes for the courses and sample exams for measuring course outcomes on the Instructor Resource Website.

What's New in This Edition?

This edition is completely revised in every detail to enhance clarity, presentation, content, examples, and exercises. The major improvements are as follows:

- Updated to Java 9, 10, and 11. Examples are improved and simplified by using the new features in Java 9, 10, 11.
- The GUI chapters are updated to JavaFX 11. The examples are revised. The user interfaces in the examples and exercises are now resizable and displayed in the center of the window.
- More examples and exercises in the data structures chapters use Lambda expressions to simplify coding.
- Both `Comparable` and `Comparator` are used to compare elements in `Heap`, `PriorityQueue`, `BST`, and `AVLTree`. This is consistent with the Java API and is more useful and flexible.
- String matching algorithms are introduced in Chapter 22.
- VideoNotes are updated.
- Provided additional exercises not printed in the book. These exercises are available for instructors only.

Please visit www.pearsonhighered.com/liang for a complete list of new features as well as correlations to the previous edition.

Pedagogical Features

The book uses the following elements to help students get the most from the material:

- The **Objectives** at the beginning of each chapter list what students should learn from the chapter. This will help them determine whether they have met the objectives after completing the chapter.
- The **Introduction** opens the discussion with a thought-provoking question to motivate the reader to delve into the chapter.
- **Key Points** highlight the important concepts covered in each section.
- **Check Points** provide review questions to help students track their progress as they read through the chapter and evaluate their learning.

- **Problems and Case Studies**, carefully chosen and presented in an easy-to-follow style, teach problem solving and programming concepts. The book uses many small, simple, and stimulating examples to demonstrate important ideas.
- The **Chapter Summary** reviews the important subjects that students should understand and remember. It helps them reinforce the key concepts they have learned in the chapter.
- **Quizzes** are accessible online, grouped by sections, for students to do self-test on programming concepts and techniques.
- **Programming Exercises** are grouped by sections to provide students with opportunities to apply the new skills they have learned on their own. The level of difficulty is rated as easy (no asterisk), moderate (*), hard (**), or challenging (***). The trick of learning programming is practice, practice, and practice. To that end, the book provides a great many exercises. Additionally, more than 200 programming exercises with solutions are provided to the instructors on the Instructor Resource Website. These exercises are not printed in the text.
- **Notes**, **Tips**, **Cautions**, and **Design Guides** are inserted throughout the text to offer valuable advice and insight on important aspects of program development.

Note
Provides additional information on the subject and reinforces important concepts.

Tip
Teaches good programming style and practice.

Caution
Helps students steer away from the pitfalls of programming errors.

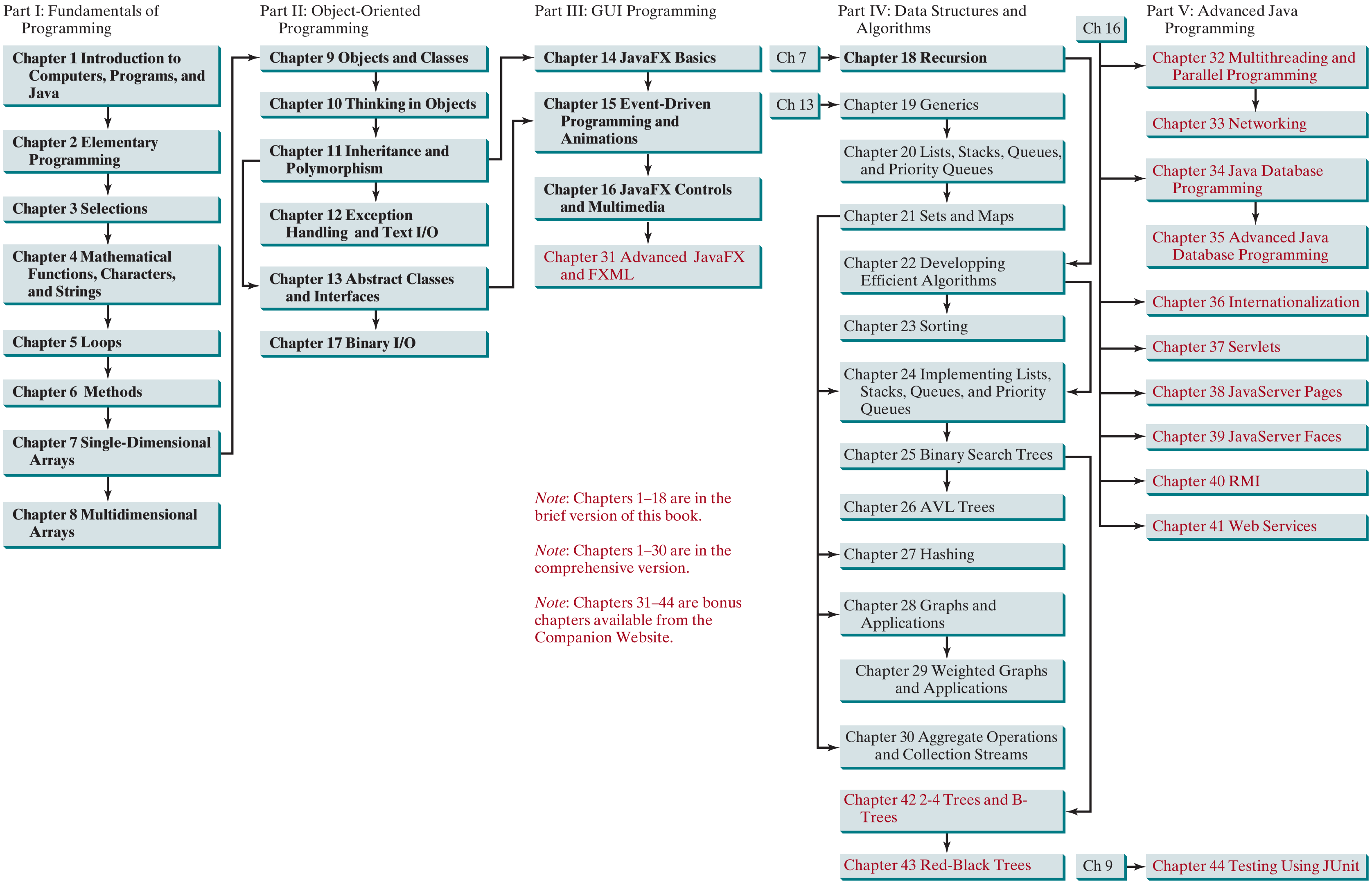
Design Guide
Provides guidelines for designing programs.

Flexible Chapter Orderings

The book is designed to provide flexible chapter orderings to enable GUI, exception handling, recursion, generics, and the Java Collections Framework to be covered earlier or later. The diagram on the next page shows the chapter dependencies.

Organization of the Book

The chapters can be grouped into five parts that, taken together, form a comprehensive introduction to Java programming, data structures and algorithms, and database and Web programming. Because knowledge is cumulative, the early chapters provide the conceptual basis for understanding programming and guide students through simple examples and exercises; subsequent chapters progressively present Java programming in detail, culminating with the development of comprehensive Java applications. The appendixes contain a mixed bag of topics, including an introduction to number systems, bitwise operations, regular expressions, and enumerated types.


Part I: Fundamentals of Programming
Chapter 1 Introduction to Computers, Programs, and Java
Chapter 2 Elementary Programming
Chapter 3 Selections
Chapter 4 Mathematical Functions, Characters, and Strings
Chapter 5 Loops
Chapter 6 Methods
Chapter 7 Single-Dimensional Arrays
Chapter 8 Multidimensional Arrays
Part II: Object-Oriented Programming
Chapter 9 Objects and Classes
Chapter 10 Thinking in Objects
Chapter 11 Inheritance and Polymorphism
Chapter 12 Exception Handling and Text I/O
Chapter 13 Abstract Classes and Interfaces
Chapter 17 Binary I/O
Part III: GUI Programming
Chapter 14 JavaFX Basics
Chapter 15 Event-Driven Programming and Animations
Chapter 16 JavaFX Controls and Multimedia
Chapter 31 Advanced JavaFX and FXML
Part IV: Data Structures and Algorithms
Ch 7
Ch 13
Chapter 18 Recursion
Chapter 19 Generics
Chapter 20 Lists, Stacks, Queues, and Priority Queues
Chapter 21 Sets and Maps
Chapter 22 Developping Efficient Algorithms
Chapter 23 Sorting
Chapter 24 Implementing Lists, Stacks, Queues, and Priority Queues
Chapter 25 Binary Search Trees
Chapter 26 AVL Trees
Chapter 27 Hashing
Chapter 28 Graphs and Applications
Chapter 29 Weighted Graphs and Applications
Chapter 30 Aggregate Operations and Collection Streams
Chapter 42 2-4 Trees and B-Trees
Chapter 43 Red-Black Trees
Ch 16
Ch 9
Part V: Advanced Java Programming
Chapter 32 Multithreading and Parallel Programming
Chapter 33 Networking
Chapter 34 Java Database Programming
Chapter 35 Advanced Java Database Programming
Chapter 36 Internationalization
Chapter 37 Servlets
Chapter 38 JavaServer Pages
Chapter 39 JavaServer Faces
Chapter 40 RMI
Chapter 41 Web Services
Chapter 44 Testing Using JUnit

Note: Chapters 1–18 are in the brief version of this book.

Note: Chapters 1–30 are in the comprehensive version.

Note: Chapters 31–44 are bonus chapters available from the Companion Website.

Part I: Fundamentals of Programming (Chapters 1–8)

The first part of the book is a stepping stone, preparing you to embark on the journey of learning Java. You will begin to learn about Java (Chapter 1) and fundamental programming techniques with primitive data types, variables, constants, assignments, expressions, and operators (Chapter 2), selection statements (Chapter 3), mathematical functions, characters, and strings (Chapter 4), loops (Chapter 5), methods (Chapter 6), and arrays (Chapters 7–8). After Chapter 7, you can jump to Chapter 18 to learn how to write recursive methods for solving inherently recursive problems.

Part II: Object-Oriented Programming (Chapters 9–13, and 17)

This part introduces object-oriented programming. Java is an object-oriented programming language that uses abstraction, encapsulation, inheritance, and polymorphism to provide great flexibility, modularity, and reusability in developing software. You will learn programming with objects and classes (Chapters 9–10), class inheritance (Chapter 11), polymorphism (Chapter 11), exception handling (Chapter 12), abstract classes (Chapter 13), and interfaces (Chapter 13). Text I/O is introduced in Chapter 12 and binary I/O is discussed in Chapter 17.

Part III: GUI Programming (Chapters 14–16 and Bonus Chapter 31)

JavaFX is a new framework for developing Java GUI programs. It is not only useful for developing GUI programs, but also an excellent pedagogical tool for learning object-oriented programming. This part introduces Java GUI programming using JavaFX in Chapters 14–16. Major topics include GUI basics (Chapter 14), container panes (Chapter 14), drawing shapes (Chapter 14), event-driven programming (Chapter 15), animations (Chapter 15), and GUI controls (Chapter 16), and playing audio and video (Chapter 16). You will learn the architecture of JavaFX GUI programming and use the controls, shapes, panes, image, and video to develop useful applications. Chapter 31 covers advanced features in JavaFX.

Part IV: Data Structures and Algorithms (Chapters 18–30 and Bonus Chapters 42–43)

This part covers the main subjects in a typical data structures and algorithms course. Chapter 18 introduces recursion to write methods for solving inherently recursive problems. Chapter 19 presents how generics can improve software reliability. Chapters 20 and 21 introduce the Java Collection Framework, which defines a set of useful API for data structures. Chapter 22 discusses measuring algorithm efficiency in order to choose an appropriate algorithm for applications. Chapter 23 describes classic sorting algorithms. You will learn how to implement several classic data structures lists, queues, and priority queues in Chapter 24. Chapters 25 and 26 introduce binary search trees and AVL trees. Chapter 27 presents hashing and implementing maps and sets using hashing. Chapters 28 and 29 introduce graph applications. Chapter 30 introduces aggregate operations for collection streams. The 2-4 trees, B-trees, and red-black trees are covered in Bonus Chapters 42–43.

Part V: Advanced Java Programming (Chapters 32-41, 44)

This part of the book is devoted to advanced Java programming. Chapter 32 treats the use of multithreading to make programs more responsive and interactive and introduces parallel programming. Chapter 33 discusses how to write programs that talk with each other from different hosts over the Internet. Chapter 34 introduces the use of Java to develop database projects. Chapter 35 delves into advanced Java database programming. Chapter 36 covers the use of internationalization support to develop projects for international audiences. Chapters 37 and 38 introduce how to use Java servlets and JavaServer Pages to generate dynamic content from Web servers. Chapter 39 introduces modern Web application development using JavaServer Faces. Chapter 40 introduces remote method invocation and Chapter 41 discusses Web services. Chapter 44 introduces testing Java programs using JUnit.

Appendixes

This part of the book covers a mixed bag of topics. Appendix A lists Java keywords. Appendix B gives tables of ASCII characters and their associated codes in decimal and in hex. Appendix C shows the operator precedence. Appendix D summarizes Java modifiers and their usage. Appendix E discusses special floating-point values. Appendix F introduces number systems and conversions among binary, decimal, and hex numbers. Finally, Appendix G introduces bitwise operations. Appendix H introduces regular expressions. Appendix I covers enumerated types.

Java Development Tools

IDE tutorials

You can use a text editor, such as the Windows Notepad or WordPad, to create Java programs and to compile and run the programs from the command window. You can also use a Java development tool, such as NetBeans or Eclipse. These tools support an integrated development environment (IDE) for developing Java programs quickly. Editing, compiling, building, executing, and debugging programs are integrated in one graphical user interface. Using these tools effectively can greatly increase your programming productivity. NetBeans and Eclipse are easy to use if you follow the tutorials. Tutorials on NetBeans and Eclipse can be found in the supplements on the Companion Website www.pearsonhighered.com/liang.

Student Resource Website

The Student Resource Website (www.pearsonhighered.com/liang) contains the following resources:

- Answers to CheckPoint questions
- Solutions to majority of even-numbered programming exercises
- Source code for the examples in the book
- Interactive quiz (organized by sections for each chapter)
- Supplements
- Debugging tips
- Video notes
- Algorithm animations
- Errata

Supplements

The text covers the essential subjects. The supplements extend the text to introduce additional topics that might be of interest to readers. The supplements are available from the Companion Website.

Instructor Resource Website

The Instructor Resource Website, accessible from www.pearsonhighered.com/liang, contains the following resources:

- Microsoft PowerPoint slides with interactive buttons to view full-color, syntax-highlighted source code and to run programs without leaving the slides.
- Solutions to majority of odd-numbered programming exercises.

- More than 200 additional programming exercises and 300 quizzes organized by chapters. These exercises and quizzes are available only to the instructors. Solutions to these exercises and quizzes are provided.
- Web-based quiz generator. (Instructors can choose chapters to generate quizzes from a large database of more than two thousand questions.)
- Sample exams. Most exams have four parts:
    - Multiple-choice questions or short-answer questions
    - Correct programming errors
    - Trace programs
    - Write programs
- Sample exams with ABET course assessment.
- Projects. In general, each project gives a description and asks students to analyze, design, and implement the project.

Some readers have requested the materials from the Instructor Resource Website. Please understand that these are for instructors only. Such requests will not be answered.

Online Practice and Assessment with MyProgrammingLab

MyProgrammingLab™

MyProgrammingLab helps students fully grasp the logic, semantics, and syntax of programming. Through practice exercises and immediate, personalized feedback, MyProgrammingLab improves the programming competence of beginning students who often struggle with the basic concepts and paradigms of popular high-level programming languages.

A self-study and homework tool, a MyProgrammingLab course consists of hundreds of small practice problems organized around the structure of this textbook. For students, the system automatically detects errors in the logic and syntax of their code submissions and offers targeted hints that enable students to figure out what went wrong—and why. For instructors, a comprehensive gradebook tracks correct and incorrect answers and stores the code inputted by students for review.

MyProgrammingLab is offered to users of this book in partnership with Turing's Craft, the makers of the CodeLab interactive programming exercise system. For a full demonstration, to see feedback from instructors and students, or to get started using MyProgrammingLab in your course, visit www.myprogramminglab.com.

Video Notes

VideoNote

We are excited about the new Video Notes feature that is found in this new edition. These videos provide additional help by presenting examples of key topics and showing how to solve problems completely from design through coding. Video Notes are available from www.pearsonhighered.com/liang.

Algorithm Animations

Animation

We have provided numerous animations for algorithms. These are valuable pedagogical tools to demonstrate how algorithms work. Algorithm animations can be accessed from the Companion Website.

Acknowledgments

I would like to thank Georgia Southern University for enabling me to teach what I write and for supporting me in writing what I teach. Teaching is the source of inspiration for continuing to improve the book. I am grateful to the instructors and students who have offered comments, suggestions, corrections, and praise. My special thanks go to Stefan Andrei of Lamar University and William Bahn of University of Colorado Colorado Springs for their help to improve the data structures part of this book.

This book has been greatly enhanced thanks to outstanding reviews for this and previous editions. The reviewers are: Elizabeth Adams (James Madison University), Syed Ahmed (North Georgia College and State University), Omar Aldawud (Illinois Institute of Technology), Stefan Andrei (Lamar University), Yang Ang (University of Wollongong, Australia), Kevin Bierre (Rochester Institute of Technology), Aaron Braskin (Mira Costa High School), David Champion (DeVry Institute), James Chegwidden (Tarrant County College), Anup Dargar (University of North Dakota), Daryl Detrick (Warren Hills Regional High School), Charles Dierbach (Towson University), Frank Ducrest (University of Louisiana at Lafayette), Erica Eddy (University of Wisconsin at Parkside), Summer Ehresman (Center Grove High School), Deena Engel (New York University), Henry A. Etlinger (Rochester Institute of Technology), James Ten Eyck (Marist College), Myers Foreman (Lamar University), Olac Fuentes (University of Texas at El Paso), Edward F. Gehringer (North Carolina State University), Harold Grossman (Clemson University), Barbara Guillot (Louisiana State University), Stuart Hansen (University of Wisconsin, Parkside), Dan Harvey (Southern Oregon University), Ron Hofman (Red River College, Canada), Stephen Hughes (Roanoke College), Vladan Jovanovic (Georgia Southern University), Deborah Kabura Kariuki (Stony Point High School), Edwin Kay (Lehigh University), Larry King (University of Texas at Dallas), Nana Kofi (Langara College, Canada), George Koutsogiannakis (Illinois Institute of Technology), Roger Kraft (Purdue University at Calumet), Norman Krumpe (Miami University), Hong Lin (DeVry Institute), Dan Lipsa (Armstrong State University), James Madison (Rensselaer Polytechnic Institute), Frank Malinowski (Darton College), Tim Margush (University of Akron), Debbie Masada (Sun Microsystems), Blayne Mayfield (Oklahoma State University), John McGrath (J.P. McGrath Consulting), Hugh McGuire (Grand Valley State), Shyamal Mitra (University of Texas at Austin), Michel Mitri (James Madison University), Kenrick Mock (University of Alaska Anchorage), Frank Murgolo (California State University, Long Beach), Jun Ni (University of Iowa), Benjamin Nystuen (University of Colorado at Colorado Springs), Maureen Opkins (CA State University, Long Beach), Gavin Osborne (University of Saskatchewan), Kevin Parker (Idaho State University), Dale Parson (Kutztown University), Mark Pendergast (Florida Gulf Coast University), Richard Povinelli (Marquette University), Roger Priebe (University of Texas at Austin), Mary Ann Pumphrey (De Anza Junior College), Pat Roth (Southern Polytechnic State University), Amr Sabry (Indiana University), Ben Setzer (Kennesaw State University), Carolyn Schauble (Colorado State University), David Scuse (University of Manitoba), Ashraf Shirani (San Jose State University), Daniel Spiegel (Kutztown University), Joslyn A. Smith (Florida Atlantic University), Lixin Tao (Pace University), Ronald F. Taylor (Wright State University), Russ Tront (Simon Fraser University), Deborah Trytten (University of Oklahoma), Michael Verdicchio (Citadel), Kent Vidrine (George Washington University), and Bahram Zartoshty (California State University at Northridge).

It is a great pleasure, honor, and privilege to work with Pearson. I would like to thank Tracy Johnson and her colleagues Marcia Horton, Demetrius Hall, Yvonne Vannatta, Kristy Alaura, Carole Snyder, Scott Disanno, Bob Engelhardt, Shylaja Gattupalli, and their colleagues for organizing, producing, and promoting this project.

As always, I am indebted to my wife, Samantha, for her love, support, and encouragement.

BRIEF CONTENTS

1 Introduction to Computers, Programs, and Java™ 1
2 Elementary Programming 33
3 Selections 77
4 Mathematical Functions, Characters, and Strings 121
5 Loops 159
6 Methods 205
7 Single-Dimensional Arrays 249
8 Multidimensional Arrays 289
9 Objects and Classes 323
10 Object-Oriented Thinking 367
11 Inheritance and Polymorphism 411
12 Exception Handling and Text I/O 453
13 Abstract Classes and Interfaces 499
14 JavaFX Basics 541
15 Event-Driven Programming and Animations 593
16 JavaFX UI Controls and Multimedia 643
17 Binary I/O 691
18 Recursion 719
19 Generics 751
20 Lists, Stacks, Queues, and Priority Queues 775
21 Sets and Maps 815
22 Developing Efficient Algorithms 839
23 Sorting 887
24 Implementing Lists, Stacks, Queues, and Priority Queues 923
25 Binary Search Trees 959
26 AVL Trees 995
27 Hashing 1015
28 Graphs and Applications 1045
29 Weighted Graphs and Applications 1091
30 Aggregate Operations for Collection Streams 1129

CHAPTER 31–44 are available from the Companion Website at www.pearsonhighered.com/liang

31 Advanced JavaFX and FXML
32 Multithreading and Parallel Programming
33 Networking
34 Java Database Programming
35 Advanced Java Database Programming
36 Internationalization
37 Servlets
38 JavaServer Pages
39 JavaServer Faces
40 RMI
41 Web Services
42 2-4 Trees and B-Trees
43 Red-Black Trees
44 Testing Using JUnit

APPENDIXES 1161
A Java Keywords and Reserved Words 1163
B The ASCII Character Set 1164
C Operator Precedence Chart 1166
D Java Modifiers 1168
E Special Floating-Point Values 1170
F Number Systems 1171
G Bitwise Operations 1175
H Regular Expressions 1176
I Enumerated Types 1182
J The Big-O, Big-Omega, and Big-Theta Notations 1187

QUICK REFERENCE 1189
INDEX 1191

CONTENTS

Chapter 1 Introduction to Computers, Programs, and Java™ 1
1.1 Introduction 2
1.2 What Is a Computer? 2
1.3 Programming Languages 7
1.4 Operating Systems 9
1.5 Java, the World Wide Web, and Beyond 10
1.6 The Java Language Specification, API, JDK, JRE, and IDE 11
1.7 A Simple Java Program 12
1.8 Creating, Compiling, and Executing a Java Program 15
1.9 Programming Style and Documentation 18
1.10 Programming Errors 19
1.11 Developing Java Programs Using NetBeans 23
1.12 Developing Java Programs Using Eclipse 26

Chapter 2 Elementary Programming 33
2.1 Introduction 34
2.2 Writing a Simple Program 34
2.3 Reading Input from the Console 37
2.4 Identifiers 40
2.5 Variables 40
2.6 Assignment Statements and Assignment Expressions 42
2.7 Named Constants 43
2.8 Naming Conventions 44
2.9 Numeric Data Types and Operations 45
2.10 Numeric Literals 48
2.11 JShell 50
2.12 Evaluating Expressions and Operator Precedence 52
2.13 Case Study: Displaying the Current Time 54
2.14 Augmented Assignment Operators 56
2.15 Increment and Decrement Operators 57
2.16 Numeric Type Conversions 58
2.17 Software Development Process 61
2.18 Case Study: Counting Monetary Units 64
2.19 Common Errors and Pitfalls 67

Chapter 3 Selections 77
3.1 Introduction 78
3.2 `boolean` Data Type, Values, and Expressions 78
3.3 `if` Statements 80
3.4 Two-Way `if-else` Statements 82
3.5 Nested `if` and Multi-Way `if-else` Statements 83
3.6 Common Errors and Pitfalls 85
3.7 Generating Random Numbers 89
3.8 Case Study: Computing Body Mass Index 91
3.9 Case Study: Computing Taxes 92
3.10 Logical Operators 95
3.11 Case Study: Determining Leap Year 99
3.12 Case Study: Lottery 100

3.13 `switch` Statements 102
3.14 Conditional Operators 105
3.15 Operator Precedence and Associativity 106
3.16 Debugging 108

Chapter 4 Mathematical Functions, Characters, and Strings 121

4.1 Introduction 122
4.2 Common Mathematical Functions 122
4.3 Character Data Type and Operations 126
4.4 The String Type 131
4.5 Case Studies 140
4.6 Formatting Console Output 146

Chapter 5 Loops 159

5.1 Introduction 160
5.2 The while Loop 160
5.3 Case Study: Guessing Numbers 163
5.4 Loop Design Strategies 166
5.5 Controlling a Loop with User Confirmation or a Sentinel Value 168
5.6 The `do-while` Loop 171
5.7 The `for` Loop 173
5.8 Which Loop to Use? 176
5.9 Nested Loops 178
5.10 Minimizing Numeric Errors 180
5.11 Case Studies 182
5.12 Keywords *`break`* and *`continue`* 186
5.13 Case Study: Checking Palindromes 189
5.14 Case Study: Displaying Prime Numbers 191

Chapter 6 Methods 205

6.1 Introduction 206
6.2 Defining a Method 206
6.3 Calling a Method 208
6.4 `void` vs. Value-Returning Methods 211
6.5 Passing Arguments by Values 213
6.6 Modularizing Code 217
6.7 Case Study: Converting Hexadecimals to Decimals 219
6.8 Overloading Methods 221
6.9 The Scope of Variables 224
6.10 Case Study: Generating Random Characters 225
6.11 Method Abstraction and Stepwise Refinement 227

Chapter 7 Single-Dimensional Arrays 249

7.1 Introduction 250
7.2 Array Basics 250
7.3 Case Study: Analyzing Numbers 257
7.4 Case Study: Deck of Cards 258
7.5 Copying Arrays 260
7.6 Passing Arrays to Methods 261
7.7 Returning an Array from a Method 264
7.8 Case Study: Counting the Occurrences of Each Letter 265
7.9 Variable-Length Argument Lists 268
7.10 Searching Arrays 269

7.11 Sorting Arrays 273
7.12 The `Arrays` Class 274
7.13 Command-Line Arguments 276

Chapter 8 Multidimensional Arrays 289
8.1 Introduction 290
8.2 Two-Dimensional Array Basics 290
8.3 Processing Two-Dimensional Arrays 293
8.4 Passing Two-Dimensional Arrays to Methods 295
8.5 Case Study: Grading a Multiple-Choice Test 296
8.6 Case Study: Finding the Closest Pair 298
8.7 Case Study: Sudoku 300
8.8 Multidimensional Arrays 303

Chapter 9 Objects and Classes 323
9.1 Introduction 324
9.2 Defining Classes for Objects 324
9.3 Example: Defining Classes and Creating Objects 326
9.4 Constructing Objects Using Constructors 331
9.5 Accessing Objects via Reference Variables 332
9.6 Using Classes from the Java Library 336
9.7 Static Variables, Constants, and Methods 339
9.8 Visibility Modifiers 344
9.9 Data Field Encapsulation 346
9.10 Passing Objects to Methods 349
9.11 Array of Objects 353
9.12 Immutable Objects and Classes 355
9.13 The Scope of Variables 357
9.14 The `this` Reference 358

Chapter 10 Object-Oriented Thinking 367
10.1 Introduction 368
10.2 Class Abstraction and Encapsulation 368
10.3 Thinking in Objects 372
10.4 Class Relationships 375
10.5 Case Study: Designing the `Course` Class 378
10.6 Case Study: Designing a Class for Stacks 380
10.7 Processing Primitive Data Type Values as Objects 382
10.8 Automatic Conversion between Primitive Types and Wrapper Class Types 386
10.9 The `BigInteger` and `BigDecimal` Classes 387
10.10 The String Class 388
10.11 The StringBuilder and StringBuffer Classes 395

Chapter 11 Inheritance and Polymorphism 411
11.1 Introduction 412
11.2 Superclasses and Subclasses 412
11.3 Using the `super` Keyword 418
11.4 Overriding Methods 421
11.5 Overriding vs. Overloading 422
11.6 The `Object` Class and Its `toString()` Method 424
11.7 Polymorphism 425
11.8 Dynamic Binding 425
11.9 Casting Objects and the `instanceof` Operator 429

11.10 The Object's equals Method 433
11.11 The ArrayList Class 434
11.12 Useful Methods for Lists 440
11.13 Case Study: A Custom Stack Class 441
11.14 The `protected` Data and Methods 442
11.15 Preventing Extending and Overriding 445

Chapter 12 Exception Handling and Text I/O 453

12.1 Introduction 454
12.2 Exception-Handling Overview 454
12.3 Exception Types 459
12.4 Declaring, Throwing, and Catching Exceptions 462
12.5 The `finally` Clause 470
12.6 When to Use Exceptions 472
12.7 Rethrowing Exceptions 472
12.8 Chained Exceptions 473
12.9 Defining Custom Exception Classes 474
12.10 The `File` Class 477
12.11 File Input and Output 480
12.12 Reading Data from the Web 487
12.13 Case Study: Web Crawler 488

Chapter 13 Abstract Classes and Interfaces 499

13.1 Introduction 500
13.2 Abstract Classes 500
13.3 Case Study: The Abstract `Number` Class 505
13.4 Case Study: `Calendar` and `GregorianCalendar` 507
13.5 Interfaces 510
13.6 The `Comparable` Interface 514
13.7 The `Cloneable` Interface 518
13.8 Interfaces vs. Abstract Classes 523
13.9 Case Study: The `Rational` Class 526
13.10 Class-Design Guidelines 531

Chapter 14 JavaFX Basics 541

14.1 Introduction 542
14.2 JavaFX vs Swing and AWT 542
14.3 The Basic Structure of a JavaFX Program 542
14.4 Panes, Groups, UI Controls, and Shapes 545
14.5 Property Binding 548
14.6 Common Properties and Methods for Nodes 551
14.7 The `Color` Class 553
14.8 The `Font` Class 554
14.9 The `Image` and `ImageView` Classes 556
14.10 Layout Panes and Groups 558
14.11 Shapes 567
14.12 Case Study: The `ClockPane` Class 580

Chapter 15 Event-Driven Programming and Animations 593

15.1 Introduction 594
15.2 Events and Event Sources 596
15.3 Registering Handlers and Handling Events 597
15.4 Inner Classes 601

15.5 Anonymous Inner Class Handlers 602
15.6 Simplifying Event Handling Using Lambda Expressions 605
15.7 Case Study: Loan Calculator 609
15.8 Mouse Events 611
15.9 Key Events 613
15.10 Listeners for Observable Objects 616
15.11 Animation 618
15.12 Case Study: Bouncing Ball 626
15.13 Case Study: US Map 630

Chapter 16 JavaFX UI Controls and Multimedia 643
16.1 Introduction 644
16.2 `Labeled` and `Label` 644
16.3 `Button` 646
16.4 `CheckBox` 648
16.5 `RadioButton` 651
16.6 `TextField` 654
16.7 `TextArea` 655
16.8 `ComboBox` 659
16.9 `ListView` 662
16.10 `ScrollBar` 665
16.11 `Slider` 668
16.12 Case Study: Developing a Tic-Tac-Toe Game 671
16.13 Video and Audio 676
16.14 Case Study: National Flags and Anthems 679

Chapter 17 Binary I/O 691
17.1 Introduction 692
17.2 How Is Text I/O Handled in Java? 692
17.3 Text I/O vs. Binary I/O 693
17.4 Binary I/O Classes 694
17.5 Case Study: Copying Files 704
17.6 Object I/O 706
17.7 Random-Access Files 711

Chapter 18 Recursion 719
18.1 Introduction 720
18.2 Case Study: Computing Factorials 720
18.3 Case Study: Computing Fibonacci Numbers 723
18.4 Problem Solving Using Recursion 726
18.5 Recursive Helper Methods 728
18.6 Case Study: Finding the Directory Size 731
18.7 Case Study: Tower of Hanoi 733
18.8 Case Study: Fractals 736
18.9 Recursion vs. Iteration 740
18.10 Tail Recursion 740

Chapter 19 Generics 751
19.1 Introduction 752
19.2 Motivations and Benefits 752
19.3 Defining Generic Classes and Interfaces 754
19.4 Generic Methods 756
19.5 Case Study: Sorting an Array of Objects 758

19.6 Raw Types and Backward Compatibility 760
19.7 Wildcard Generic Types 761
19.8 Erasure and Restrictions on Generics 764
19.9 Case Study: Generic Matrix Class 766

Chapter 20 Lists, Stacks, Queues, and Priority Queues 775
20.1 Introduction 776
20.2 Collections 776
20.3 Iterators 780
20.4 Using the `forEach` Method 782
20.5 Lists 783
20.6 The `Comparator` Interface 787
20.7 Static Methods for Lists and Collections 792
20.8 Case Study: Bouncing Balls 795
20.9 `Vector` and `Stack` Classes 798
20.10 Queues and Priority Queues 800
20.11 Case Study: Evaluating Expressions 803

Chapter 21 Sets and Maps 815
21.1 Introduction 816
21.2 Sets 816
21.3 Comparing the Performance of Sets and Lists 824
21.4 Case Study: Counting Keywords 827
21.5 Maps 828
21.6 Case Study: Occurrences of Words 833
21.7 Singleton and Unmodifiable Collections and Maps 835

Chapter 22 Developing Efficient Algorithms 839
22.1 Introduction 840
22.2 Measuring Algorithm Efficiency Using Big *O* Notation 840
22.3 Examples: Determining Big *O* 842
22.4 Analyzing Algorithm Time Complexity 846
22.5 Finding Fibonacci Numbers Using Dynamic Programming 849
22.6 Finding Greatest Common Divisors Using Euclid's Algorithm 851
22.7 Efficient Algorithms for Finding Prime Numbers 855
22.8 Finding the Closest Pair of Points Using Divide-and-Conquer 861
22.9 Solving the Eight Queens Problem Using Backtracking 864
22.10 Computational Geometry: Finding a Convex Hull 867
22.11 String Matching 869

Chapter 23 Sorting 887
23.1 Introduction 888
23.2 Insertion Sort 888
23.3 Bubble Sort 890
23.4 Merge Sort 892
23.5 Quick Sort 896
23.6 Heap Sort 900
23.7 Bucket and Radix Sorts 907
23.8 External Sort 909

Chapter 24 Implementing Lists, Stacks, Queues, and Priority Queues 923
24.1 Introduction 924
24.2 Common Operations for Lists 924
24.3 Array Lists 928
24.4 Linked Lists 935
24.5 Stacks and Queues 949
24.6 Priority Queues 953

Chapter 25 Binary Search Trees 959
25.1 Introduction 960
25.2 Binary Search Trees Basics 960
25.3 Representing Binary Search Trees 961
25.4 Searching for an Element 962
25.5 Inserting an Element into a BST 962
25.6 Tree Traversal 963
25.7 The BST Class 965
25.8 Deleting Elements from a BST 974
25.9 Tree Visualization and MVC 980
25.10 Iterators 983
25.11 Case Study: Data Compression 985

Chapter 26 AVL Trees 995
26.1 Introduction 996
26.2 Rebalancing Trees 996
26.3 Designing Classes for AVL Trees 999
26.4 Overriding the `insert` Method 1000
26.5 Implementing Rotations 1001
26.6 Implementing the `delete` Method 1002
26.7 The `AVLTree` Class 1002
26.8 Testing the `AVLTree` Class 1008
26.9 AVL Tree Time Complexity Analysis 1011

Chapter 27 Hashing 1015
27.1 Introduction 1016
27.2 What Is Hashing? 1016
27.3 Hash Functions and Hash Codes 1017
27.4 Handling Collisions Using Open Addressing 1019
27.5 Handling Collisions Using Separate Chaining 1023
27.6 Load Factor and Rehashing 1025
27.7 Implementing a Map Using Hashing 1025
27.8 Implementing Set Using Hashing 1034

Chapter 28 Graphs and Applications 1045
28.1 Introduction 1046
28.2 Basic Graph Terminologies 1047
28.3 Representing Graphs 1048
28.4 Modeling Graphs 1054
28.5 Graph Visualization 1064
28.6 Graph Traversals 1067

28.7 Depth-First Search (DFS) 1068
28.8 Case Study: The Connected Circles Problem 1072
28.9 Breadth-First Search (BFS) 1074
28.10 Case Study: The Nine Tails Problem 1077

Chapter 29 Weighted Graphs and Applications 1091
29.1 Introduction 1092
29.2 Representing Weighted Graphs 1093
29.3 The `WeightedGraph` Class 1095
29.4 Minimum Spanning Trees 1103
29.5 Finding Shortest Paths 1109
29.6 Case Study: The Weighted Nine Tails Problem 1118

Chapter 30 Aggregate Operations for Collection Streams 1129
30.1 Introduction 1130
30.2 Stream Pipelines 1130
30.3 `IntStream`, `LongStream`, and `DoubleStream` 1136
30.4 Parallel Streams 1139
30.5 Stream Reduction Using the `reduce` Method 1141
30.6 Stream Reduction Using the `collect` Method 1144
30.7 Grouping Elements Using the `groupingby` Collector 1147
30.8 Case Studies 1150

Chapter 31–44 are available from the Companion Website at www.pearsonhighered.com/liang

Chapter 31 Advanced JavaFX and FXML
Chapter 32 Multithreading and Parallel Programming
Chapter 33 Networking
Chapter 34 Java Database Programming
Chapter 35 Advanced Database Programming
Chapter 36 Internationalization
Chapter 37 Servlets
Chapter 38 JavaServer Pages
Chapter 39 JavaServer Faces
Chapter 40 RMI
Chapter 41 Web Services

Chapter 42 2-4 Trees and B-Trees
Chapter 43 Red-Black Trees
Chapter 44 Testing Using JUnit

APPENDIXES 1161
Appendix A Java Keywords and Reserved Words 1163
Appendix B The ASCII Character Set 1164
Appendix C Operator Precedence Chart 1166
Appendix D Java Modifiers 1168
Appendix E Special Floating-Point Values 1170
Appendix F Number Systems 1171
Appendix G Bitwise Operations 1175
Appendix H Regular Expressions 1176
Appendix I Enumerated Types 1182
Appendix J The Big-O, Big-Omega, and Big-Theta Notations 1187

QUICK REFERENCE 1189
INDEX 1191

VideoNotes

Locations of VideoNotes

http://www.pearsonhighered.com/liang

Chapter 1 Introduction to Computers, Programs, and Java™ 1
Your first Java program 12
Compile and Run a Java Program 17
NetBeans brief tutorial 23
Eclipse brief tutorial 26

Chapter 2 Elementary Programming 33
Obtain Input 37
Use operators / and % 54
Software development process 61
Compute loan payments 62
Compute BMI 73

Chapter 3 Selections 77
Program addition quiz 79
Program subtraction quiz 89
Use multi-way `if-else` statements 92
Sort three integers 112
Check point location 114

Chapter 4 Mathematical Functions, Characters, and Strings 121
Introduce Math functions 122
Introduce strings and objects 131
Convert hex to decimal 143
Compute great circle distance 152
Convert hex to binary 154

Chapter 5 Loops 159
Use while loop 160
Guess a number 163
Multiple subtraction quiz 166
Use do-while loop 171
Minimize numeric errors 180
Display loan schedule 197
Sum a series 198

Chapter 6 Methods 205
Define/invoke `max` method 208
Use `void` method 211
Modularize code 217
Stepwise refinement 227
Reverse an integer 236
Estimate π 240

Chapter 7 Single-Dimensional Arrays 249
Random shuffling 254
Deck of cards 258
Selection sort 273
Command-line arguments 277
Coupon collector's problem 284
Consecutive four 286

Chapter 8 Multidimensional Arrays 289
Find the row with the largest sum 294
Grade multiple-choice test 296
Sudoku 300
Multiply two matrices 309
Even number of 1s 316

Chapter 9 Objects and Classes 323
Define classes and create objects 324
Static vs. instance 339
Data field encapsulation 346
Immutable objects and this keyword 355
The this keyword 358
The Fan class 364

Chapter 10 Object-Oriented Thinking 367
the `Loan` class 369
The `BMI` class 372
The `StackOfIntegers` class 380
Process large numbers 387
The `String` class 388
The `MyPoint` class 403

Chapter 11 Inheritance and Polymorphism 411
Geometric class hierarchy 412
Polymorphism and dynamic binding demo 426
New `Account` class 448

Chapter 12 Exception Handling and Text I/O 453
Exception-handling advantages 454
Create custom exception classes 474
Write and read data 480
HexFormatException 493

Chapter 13 Abstract Classes and Interfaces 499
Abstract GeometricObject class 500
Calendar and GregorianCalendar classes 507
The concept of interface 510

Chapter 14 JavaFX Basics 541
Getting started with JavaFX 542
Understand property binding 548
Use `Image` and `ImageView` 556
Use layout panes 558
Use shapes 567
Display a tic-tac-toe board 586
Display a bar chart 588

Chapter 15 Event-Driven Programming and Animations 593
  Handler and its registration 600
  Anonymous handler 603
  Move message using the mouse 612
  Animate a rising flag 618
  Flashing text 624
  Simple calculator 634
  Check mouse-point location 636
  Display a running fan 639

Chapter 16 JavaFX UI Controls and Multimedia 643
  Use ListView 662
  Use `Slider` 668
  Tic-Tac-Toe 671
  Use Media, MediaPlayer, and MediaView 676
  Use radio buttons and text fields 683
  Set fonts 685

Chapter 17 Binary I/O 691
  Copy file 704
  Object I/O 706
  Split a large file 716

Chapter 18 Recursion 719
  Binary search 730
  Directory size 731
  Search a string in a directory 747
  Recursive tree 750

Animations

Chapter 7 Single-Dimensional Arrays 249
  linear search animation on Companion Website 270
  binary search animation on Companion Website 270
  selection sort animation on Companion Website 273

Chapter 8 Multidimensional Arrays 289
  closest-pair animation on the Companion Website 298

Chapter 22 Developing Efficient Algorithms 839
  binary search animation on the Companion Website 846
  selection sort animation on the Companion Website 846
  closest-pair animation on Companion Website 861
  Eight Queens animation on the Companion Website 864
  convex hull animation on the Companion Website 867

Chapter 23 Sorting 887
  insertion-sort animation on Companion Website 888
  bubble sort animation on the Companion Website 890
  merge animation on Companion Website 894
  partition animation on Companion Website 898
  radix sort on Companion Website 908

Chapter 24 Implementing Lists, Stacks, Queues, and Priority Queues 923
  list animation on Companion Website 924
  stack and queue animation on Companion Website 950

Chapter 25 Binary Search Trees 959
  BST animation on Companion Website 960

Chapter 26 AVL Trees 995
  AVL tree animation on Companion Website 996

Chapter 27 Hashing 1015
  linear probing animation on Companion Website 1020
  quadratic probing animation on Companion Website 1021
  double hashing animation on Companion Website 1022
  separate chaining animation on Companion Website 1025

Chapter 28 Graphs and Applications 1045
  graph learning tool on Companion Website 1048
  U.S. Map Search 1070

Chapter 29 Weighted Graphs and Applications 1091
  weighted graph learning tool on Companion Website 1092

CHAPTER

1

INTRODUCTION TO COMPUTERS, PROGRAMS, AND JAVA™

Objectives

- To understand computer basics, programs, and operating systems (§§1.2–1.4).
- To describe the relationship between Java and the World Wide Web (§1.5).
- To understand the meaning of Java language specification, API, JDK™, JRE™, and IDE (§1.6).
- To write a simple Java program (§1.7).
- To display output on the console (§1.7).
- To explain the basic syntax of a Java program (§1.7).
- To create, compile, and run Java programs (§1.8).
- To use sound Java programming style and document programs properly (§1.9).
- To explain the differences between syntax errors, runtime errors, and logic errors (§1.10).
- To develop Java programs using NetBeans™ (§1.11).
- To develop Java programs using Eclipse™ (§1.12).

1.1 Introduction

Key Point

The central theme of this book is to learn how to solve problems by writing a program.

what is programming?
programming
program

This book is about programming. So, what is programming? The term *programming* means to create (or develop) software, which is also called a *program*. In basic terms, software contains instructions that tell a computer—or a computerized device—what to do.

Software is all around you, even in devices you might not think would need it. Of course, you expect to find and use software on a personal computer, but software also plays a role in running airplanes, cars, cell phones, and even toasters. On a personal computer, you use word processors to write documents, web browsers to explore the Internet, and e-mail programs to send and receive messages. These programs are all examples of software. Software developers create software with the help of powerful tools called *programming languages.*

This book teaches you how to create programs by using the Java programming language. There are many programming languages, some of which are decades old. Each language was invented for a specific purpose—to build on the strengths of a previous language, for example, or to give the programmer a new and unique set of tools. Knowing there are so many programming languages available, it would be natural for you to wonder which one is best. However, in truth, there is no "best" language. Each one has its own strengths and weaknesses. Experienced programmers know one language might work well in some situations, whereas a different language may be more appropriate in other situations. For this reason, seasoned programmers try to master as many different programming languages as they can, giving them access to a vast arsenal of software-development tools.

If you learn to program using one language, you should find it easy to pick up other languages. The key is to learn how to solve problems using a programming approach. That is the main theme of this book.

You are about to begin an exciting journey: learning how to program. At the outset, it is helpful to review computer basics, programs, and operating systems (OSs). If you are already familiar with such terms as central processing unit (CPU), memory, disks, operating systems, and programming languages, you may skip Sections 1.2–1.4.

1.2 What Is a Computer?

Key Point

A computer is an electronic device that stores and processes data.

hardware
software

A computer includes both *hardware* and *software*. In general, hardware comprises the visible, physical elements of the computer, and software provides the invisible instructions that control the hardware and make it perform specific tasks. Knowing computer hardware isn't essential to learning a programming language, but it can help you better understand the effects that a program's instructions have on the computer and its components. This section introduces computer hardware components and their functions.

A computer consists of the following major hardware components (see Figure 1.1):

- A central processing unit (CPU)
- Memory (main memory)
- Storage devices (such as disks and CDs)
- Input devices (such as the mouse and the keyboard)
- Output devices (such as monitors and printers)
- Communication devices (such as modems and network interface cards (NIC))

bus

A computer's components are interconnected by a subsystem called a *bus*. You can think of a bus as a sort of system of roads running among the computer's components; data and power travel along the bus from one part of the computer to another. In personal computers,

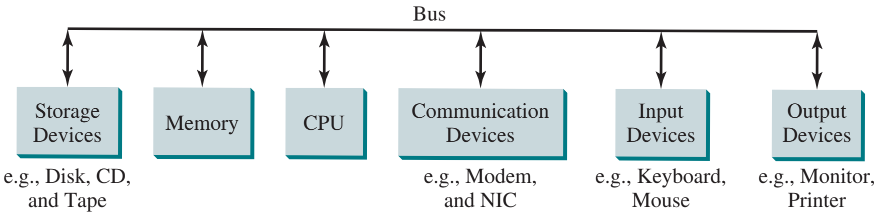


FIGURE 1.1 A computer consists of a CPU, memory, storage devices, input devices, output devices, and communication devices.

the bus is built into the computer's *motherboard*, which is a circuit case that connects all of the parts of a computer together. motherboard

1.2.1 Central Processing Unit

The *central processing unit (CPU)* is the computer's brain. It retrieves instructions from the memory and executes them. The CPU usually has two components: a *control unit* and an *arithmetic/logic unit.* The control unit controls and coordinates the actions of the other components. The arithmetic/logic unit performs numeric operations (addition, subtraction, multiplication, and division) and logical operations (comparisons). CPU

Today's CPUs are built on small silicon semiconductor chips that contain millions of tiny electric switches, called *transistors*, for processing information.

Every computer has an internal clock that emits electronic pulses at a constant rate. These pulses are used to control and synchronize the pace of operations. A higher clock *speed* enables more instructions to be executed in a given period of time. The unit of measurement of clock speed is the *hertz* (*Hz*), with 1 Hz equaling 1 pulse per second. In the 1990s, computers measured clock speed in *megahertz* (*MHz*, i.e., 1 million pulses per second), but CPU speed has been improving continuously; the clock speed of a computer is now usually stated in *gigahertz* (*GHz,* i.e., 1 billion pulses per second). Intel's newest processors run at about 3 GHz. speed hertz megahertz gigahertz

CPUs were originally developed with only one core. The *core* is the part of the processor that performs the reading and executing of instructions. In order to increase the CPU processing power, chip manufacturers are now producing CPUs that contain multiple cores. A multicore CPU is a single component with two or more independent cores. Today's consumer computers typically have two, four, and even eight separate cores. Soon, CPUs with dozens or even hundreds of cores will be affordable. core

1.2.2 Bits and Bytes

Before we discuss memory, let's look at how information (data and programs) are stored in a computer.

A computer is really nothing more than a series of switches. Each switch exists in two states: on or off. Storing information in a computer is simply a matter of setting a sequence of switches on or off. If the switch is on, its value is 1. If the switch is off, its value is 0. These 0s and 1s are interpreted as digits in the binary number system and are called *bits* (binary digits). bits

The minimum storage unit in a computer is a *byte*. A byte is composed of eight bits. A small number such as **3** can be stored as a single byte. To store a number that cannot fit into a single byte, the computer uses several bytes. byte

Data of various kinds, such as numbers and characters, are encoded as a series of bytes. As a programmer, you don't need to worry about the encoding and decoding of data, which the computer system performs automatically, based on the encoding scheme. An *encoding scheme* is a set of rules that govern how a computer translates characters and numbers into data with which the computer can actually work. Most schemes translate each character into encoding scheme a predetermined string of bits. In the popular ASCII encoding scheme, for example, the character **`C`** is represented as **`01000011`** in 1 byte.

A computer's storage capacity is measured in bytes and multiples of the byte, as follows:

kilobyte (KB)
megabyte (MB)
gigabyte (GB)
terabyte (TB)

- A *kilobyte (KB)* is about 1,000 bytes.
- A *megabyte (MB)* is about 1 million bytes.
- A *gigabyte (GB)* is about 1 billion bytes.
- A *terabyte (TB)* is about 1 trillion bytes.

A typical one-page word document might take 20 KB. Therefore, 1 MB can store 50 pages of documents, and 1 GB can store 50,000 pages of documents. A typical two-hour high-resolution movie might take 8 GB, so it would require 160 GB to store 20 movies.

1.2.3 Memory

memory

A computer's *memory* consists of an ordered sequence of bytes for storing programs as well as data with which the program is working. You can think of memory as the computer's work area for executing a program. A program and its data must be moved into the computer's memory before they can be executed by the CPU.

unique address
RAM

Every byte in the memory has a *unique address*, as shown in Figure 1.2. The address is used to locate the byte for storing and retrieving the data. Since the bytes in the memory can be accessed in any order, the memory is also referred to as *random-access memory (RAM)*.

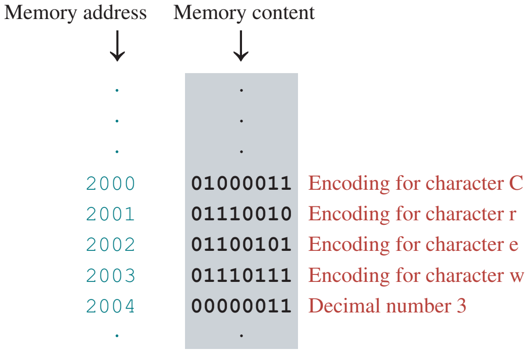


FIGURE 1.2 Memory stores data and program instructions in uniquely addressed memory locations.

Today's personal computers usually have at least 4 GB of RAM, but they more commonly have 8 to 32 GB installed. Generally speaking, the more RAM a computer has, the faster it can operate, but there are limits to this simple rule of thumb.

A memory byte is never empty, but its initial content may be meaningless to your program. The current content of a memory byte is lost whenever new information is placed in it.

Like the CPU, memory is built on silicon semiconductor chips that have millions of transistors embedded on their surface. Compared to CPU chips, memory chips are less complicated, slower, and less expensive.

1.2.4 Storage Devices

storage devices

A computer's memory (RAM) is a volatile form of data storage: Any information that has been saved in memory is lost when the system's power is turned off. Programs and data are permanently stored on *storage devices* and are moved, when the computer actually uses them, to memory, which operates at much faster speeds than permanent storage devices can.

There are four main types of storage devices:

- Magnetic disk drives
- Optical disc drives (CD and DVD)
- Universal serial bus (USB) flash drives
- Cloud storage

Drives are devices for operating a medium, such as disks and CDs. A storage medium physically stores data and program instructions. The drive reads data from the medium and writes data onto the medium.

drive

Disks

A computer usually has at least one hard disk drive. *Hard disks* are used for permanently storing data and programs. Newer computers have hard disks that can store from 1 terabyte of data to 4 terabytes of data. Hard disk drives are usually encased inside the computer, but removable hard disks are also available.

hard disk

CDs and DVDs

CD stands for compact disc. There are three types of CDs: CD-ROM, CD-R, and CD-RW. A CD-ROM is a prepressed disc. It was popular for distributing software, music, and video. Software, music, and video are now increasingly distributed on the Internet without using CDs. A *CD-R* (CD-Recordable) is a write-once medium. It can be used to record data once and read any number of times. A *CD-RW* (CD-ReWritable) can be used like a hard disk; that is, you can write data onto the disc, then overwrite that data with new data. A single CD can hold up to 700 MB.

CD-ROM
CD-R
CD-RW

DVD stands for digital versatile disc or digital video disc. DVDs and CDs look alike, and you can use either to store data. A DVD can hold more information than a CD; a standard DVD's storage capacity is 4.7 GB. There are two types of DVDs: DVD-R (Recordable) and DVD-RW (ReWritable).

DVD

USB Flash Drives

Universal serial bus (USB) connectors allow the user to attach many kinds of peripheral devices to the computer. You can use an USB to connect a printer, digital camera, mouse, external hard disk drive, and other devices to the computer.

An USB *flash drive* is a device for storing and transporting data. A flash drive is small—about the size of a pack of gum. It acts like a portable hard drive that can be plugged into your computer's USB port. USB flash drives are currently available with up to 256 GB storage capacity.

Cloud Storage

Storing data on the cloud is becoming popular. Many companies provide cloud service on the Internet. For example, you can store Microsoft Office documents in Google Docs. Google Docs can be accessed from docs.google.com on the Chrome browser. The documents can be easily shared with others. Microsoft OneDrive is provided free to Windows user for storing files. The data stored in the cloud can be accessed from any device on the Internet.

1.2.5 Input and Output Devices

Input and output devices let the user communicate with the computer. The most common input devices are the *keyboard* and *mouse*. The most common output devices are *monitors* and *printers*.

The Keyboard

A keyboard is a device for entering input. Compact keyboards are available without a numeric keypad.

function key

Function keys are located across the top of the keyboard and are prefaced with the letter *F*. Their functions depend on the software currently being used.

modifier key

A *modifier key* is a special key (such as the *Shift*, *Alt*, and *Ctrl* keys) that modifies the normal action of another key when the two are pressed simultaneously.

numeric keypad

The *numeric keypad*, located on the right side of most keyboards, is a separate set of keys styled like a calculator to use for quickly entering numbers.

arrow keys

Arrow keys, located between the main keypad and the numeric keypad, are used to move the mouse pointer up, down, left, and right on the screen in many kinds of programs.

Insert key
Delete key
Page Up key
Page Down key

The *Insert*, *Delete*, *Page Up*, and *Page Down keys* are used in word processing and other programs for inserting text and objects, deleting text and objects, and moving up or down through a document one screen at a time.

The Mouse

A *mouse* is a pointing device. It is used to move a graphical pointer (usually in the shape of an arrow) called a *cursor* around the screen, or to click on-screen objects (such as a button) to trigger them to perform an action.

The Monitor

The *monitor* displays information (text and graphics). The screen resolution and dot pitch determine the quality of the display.

screen resolution
pixels

The *screen resolution* specifies the number of pixels in horizontal and vertical dimensions of the display device. *Pixels* (short for "picture elements") are tiny dots that form an image on the screen. A common resolution for a 17-inch screen, for example, is 1,024 pixels wide and 768 pixels high. The resolution can be set manually. The higher the resolution, the sharper and clearer the image is.

dot pitch

The *dot pitch* is the amount of space between pixels, measured in millimeters. The smaller the dot pitch, the sharper is the display.

Touchscreens

The cellphones, tablets, appliances, electronic voting machines, as well as some computers use touchscreens. A touchscreen is integrated with a monitor to enable users to enter input and control the display using a finger.

1.2.6 Communication Devices

Computers can be networked through communication devices, such as a dial-up modem (*mo*dulator/*dem*odulator), a digital subscriber line (DSL) or cable modem, a wired network interface card, or a wireless adapter.

dial-up modem

- A *dial-up modem* uses a phone line to dial a phone number to connect to the Internet and can transfer data at a speed up to 56,000 bps (bits per second).

digital subscriber line (DSL)

- A *digital subscriber line (DSL)* connection also uses a standard phone line, but it can transfer data 20 times faster than a standard dial-up modem. Dial-up modem was used in the 90s and is now replaced by DSL and cable modem.

cable modem

- A *cable modem* uses the cable line maintained by the cable company and is generally faster than DSL.

network interface card (NIC)
local area network (LAN)

- A *network interface card (NIC)* is a device that connects a computer to a *local area network (LAN)*. LANs are commonly used to connect computers within a limited area such as a school, a home, and an office. A high-speed NIC called *1000BaseT* can transfer data at 1,000 million bits per second (mbps).

million bits per second (mbps)

- Wi-Fi, a special type of wireless networking, is common in homes, businesses, and schools to connect computers, phones, tablets, and printers to the Internet without the need for a physical wired connection.

Wi-Fi

Note

Answers to the CheckPoint questions are available at **www.pearsonhighered.com/liang**. Choose this book and click Companion Website to select CheckPoint.

Check Point

1.2.1 What are hardware and software?
1.2.2 List the five major hardware components of a computer.
1.2.3 What does the acronym CPU stand for? What unit is used to measure CPU speed?
1.2.4 What is a bit? What is a byte?
1.2.5 What is memory for? What does RAM stand for? Why is memory called RAM?
1.2.6 What unit is used to measure memory size? What unit is used to measure disk size?
1.2.7 What is the primary difference between memory and a storage device?

1.3 Programming Languages

Computer programs, known as software, are instructions that tell a computer what to do.

Key Point

Computers do not understand human languages, so programs must be written in a language a computer can use. There are hundreds of programming languages, and they were developed to make the programming process easier for people. However, all programs must be converted into the instructions the computer can execute.

1.3.1 Machine Language

A computer's native language, which differs among different types of computers, is its *machine language*—a set of built-in primitive instructions. These instructions are in the form of binary code, so if you want to give a computer an instruction in its native language, you have to enter the instruction as binary code. For example, to add two numbers, you might have to write an instruction in binary code as follows:

machine language

```
1101101010011010
```

1.3.2 Assembly Language

Programming in machine language is a tedious process. Moreover, programs written in machine language are very difficult to read and modify. For this reason, *assembly language* was created in the early days of computing as an alternative to machine languages. Assembly language uses a short descriptive word, known as a *mnemonic*, to represent each of the machine-language instructions. For example, the mnemonic `add` typically means to add numbers, and `sub` means to subtract numbers. To add the numbers `2` and `3` and get the result, you might write an instruction in assembly code as follows:

assembly language

```
add 2, 3, result
```

Assembly languages were developed to make programming easier. However, because the computer cannot execute assembly language, another program—called an *assembler*—is used to translate assembly-language programs into machine code, as shown in Figure 1.3.

assembler

Writing code in assembly language is easier than in machine language. However, it is still tedious to write code in assembly language. An instruction in assembly language essentially

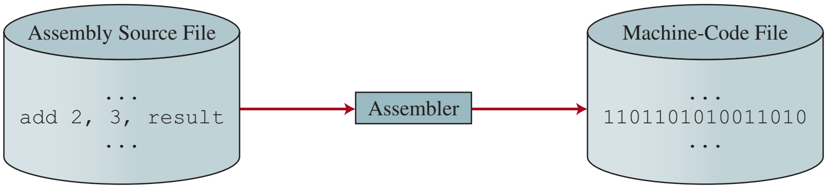


Figure 1.3 An assembler translates assembly-language instructions into machine code.

low-level language

corresponds to an instruction in machine code. Writing in assembly language requires that you know how the CPU works. Assembly language is referred to as a *low-level language*, because assembly language is close in nature to machine language and is machine dependent.

1.3.3 High-Level Language

high-level language

statement

In the 1950s, a new generation of programming languages known as *high-level languages* emerged. They are platform independent, which means that you can write a program in a high-level language and run it in different types of machines. High-level languages are similar to English and easy to learn and use. The instructions in a high-level programming language are called *statements*. Here, for example, is a high-level language statement that computes the area of a circle with a radius of **5**:

```
area = 5 * 5 * 3.14159;
```

There are many high-level programming languages, and each was designed for a specific purpose. Table 1.1 lists some popular ones.

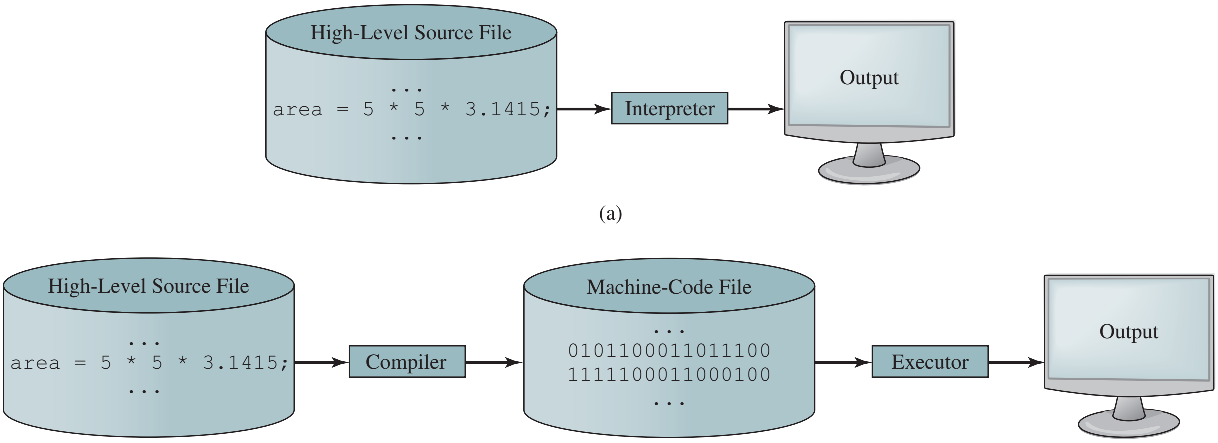
Table 1.1 Popular High-Level Programming Languages

Language	*Description*
Ada	Named for Ada Lovelace, who worked on mechanical general-purpose computers. Developed for the Department of Defense and used mainly in defense projects.
BASIC	Beginner's All-purpose Symbolic Instruction Code. Designed to be learned and used easily by beginners.
C	Developed at Bell Laboratories. Combines the power of an assembly language with the ease of use and portability of a high-level language.
C++	An object-oriented language, based on C
C#	Pronounced "C Sharp." An object-oriented programming language developed by Microsoft.
COBOL	COmmon Business Oriented Language. Used for business applications.
FORTRAN	FORmula TRANslation. Popular for scientific and mathematical applications.
Java	Developed by Sun Microsystems, now part of Oracle. An object-oriented programming language, widely used for developing platform-independent Internet applications.
JavaScript	A Web programming language developed by Netscape
Pascal	Named for Blaise Pascal, who pioneered calculating machines in the seventeenth century. A simple, structured, general-purpose language primarily for teaching programming.
Python	A simple general-purpose scripting language good for writing short programs.
Visual Basic	Visual Basic was developed by Microsoft. Enables the programmers to rapidly develop Windows-based applications.

source program
source code
interpreter
compiler

A program written in a high-level language is called a *source program* or *source code*. Because a computer cannot execute a source program, a source program must be translated into machine code for execution. The translation can be done using another programming tool called an *interpreter* or a *compiler*.

- An interpreter reads one statement from the source code, translates it to the machine code or virtual machine code, then executes it right away, as shown in Figure 1.4a. Note a statement from the source code may be translated into several machine instructions.
- A compiler translates the entire source code into a machine-code file, and the machine-code file is then executed, as shown in Figure 1.4b.


High-Level Source File
...
area = 5 * 5 * 3.1415;
...
Interpreter
Output
High-Level Source File
...
area = 5 * 5 * 3.1415;
...
Compiler
Machine-Code File
...
0101100011011100
1111100011000100
...
Executor
Output

(a)

(b)

Figure 1.4 (a) An interpreter translates and executes a program one statement at a time. (b) A compiler translates the entire source program into a machine-language file for execution.

Check Point

1.3.1 What language does the CPU understand?
1.3.2 What is an assembly language? What is an assembler?
1.3.3 What is a high-level programming language? What is a source program?
1.3.4 What is an interpreter? What is a compiler?
1.3.5 What is the difference between an interpreted language and a compiled language?

1.4 Operating Systems

Key Point

The operating system (OS) *is the most important program that runs on a computer. The OS manages and controls a computer's activities.*

operating system (OS)

The popular *operating systems* for general-purpose computers are Microsoft Windows, Mac OS, and Linux. Application programs, such as a web browser or a word processor, cannot run unless an operating system is installed and running on the computer. Figure 1.5 shows the interrelationship of hardware, operating system, application software, and the user.

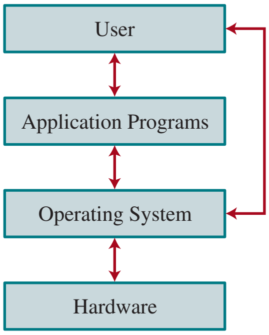

User
Application Programs
Operating System
Hardware

Figure 1.5 Users and applications access the computer's hardware via the operating system.

The major tasks of an operating system are as follows:

- Controlling and monitoring system activities
- Allocating and assigning system resources
- Scheduling operations

1.4.1 Controlling and Monitoring System Activities

Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the monitor, keeping track of files and folders on storage devices, and controlling peripheral devices such as disk drives and printers. An operating system must also ensure different programs and users working at the same time do not interfere with each other. In addition, the OS is responsible for security, ensuring unauthorized users and programs are not allowed to access the system.

1.4.2 Allocating and Assigning System Resources

The operating system is responsible for determining what computer resources a program needs (such as CPU time, memory space, disks, and input and output devices) and for allocating and assigning them to run the program.

1.4.3 Scheduling Operations

multiprogramming
multithreading
multiprocessing

The OS is responsible for scheduling programs' activities to make efficient use of system resources. Many of today's operating systems support techniques such as *multiprogramming*, *multithreading*, and *multiprocessing* to increase system performance.

Multiprogramming allows multiple programs such as Microsoft Word, E-mail, and web browser to run simultaneously by sharing the same CPU. The CPU is much faster than the computer's other components. As a result, it is idle most of the time—for example, while waiting for data to be transferred from a disk or waiting for other system resources to respond. A multiprogramming OS takes advantage of this situation by allowing multiple programs to use the CPU when it would otherwise be idle. For example, multiprogramming enables you to use a word processor to edit a file at the same time as your web browser is downloading a file.

Multithreading allows a single program to execute multiple tasks at the same time. For instance, a word-processing program allows users to simultaneously edit text and save it to a disk. In this example, editing and saving are two tasks within the same program. These two tasks may run concurrently.

Multiprocessing is similar to multithreading. The difference is that multithreading is for running multithreads concurrently within one program, but multiprocessing is for running multiple programs concurrently using multiple processors.

Check Point

1.4.1 What is an operating system? List some popular operating systems.
1.4.2 What are the major responsibilities of an operating system?
1.4.3 What are multiprogramming, multithreading, and multiprocessing?

1.5 Java, the World Wide Web, and Beyond

Key Point

Java is a powerful and versatile programming language for developing software running on mobile devices, desktop computers, and servers.

This book introduces Java programming. Java was developed by a team led by James Gosling at Sun Microsystems. Sun Microsystems was purchased by Oracle in 2010. Originally called *Oak,* Java was designed in 1991 for use in embedded chips in consumer electronic appliances.

In 1995, renamed *Java*, it was redesigned for developing web applications. For the history of Java, see www.java.com/en/javahistory/index.jsp.

Java has become enormously popular. Its rapid rise and wide acceptance can be traced to its design characteristics, particularly its promise that you can write a program once and run it anywhere. As stated by its designer, Java is *simple*, *object oriented*, *distributed*, *interpreted*, *robust*, *secure*, *architecture neutral*, *portable*, *high performance*, *multithreaded*, and *dynamic*. For the anatomy of Java characteristics, see liveexample.pearsoncmg.com/etc/JavaCharacteristics.pdf.

Java is a full-featured, general-purpose programming language that can be used to develop robust mission-critical applications. It is employed not only on desktop computers, but also on servers and mobile devices. Today, more than 3 billion devices run Java. Most major companies use Java in some applications. Most server-side applications were developed using Java. Java was used to develop the code to communicate with and control the robotic rover on Mars. The software for Android cell phones is developed using Java.

Java initially became attractive because Java programs can run from a web browser. Such programs are called *applets*. Today applets are no longer allowed to run from a Web browser due to security issues. Java, however, is now very popular for developing applications on web servers. These applications process data, perform computations, and generate dynamic webpages. Many commercial Websites are developed using Java on the backend.

Check Point

1.5.1 Who invented Java? Which company owns Java now?

1.5.2 What is a Java applet?

1.5.3 What programming language does Android use?

1.6 The Java Language Specification, API, JDK, JRE, and IDE

Key Point

Java syntax is defined in the Java language specification, and the Java library is defined in the Java application program interface (API). The JDK is the software for compiling and running Java programs. An IDE is an integrated development environment for rapidly developing programs.

Computer languages have strict rules of usage. If you do not follow the rules when writing a program, the computer will not be able to understand it. The Java language specification and the Java API define the Java standards.

Java language specification

The *Java language specification* is a technical definition of the Java programming language's syntax and semantics. You can find the complete Java language specification at docs.oracle.com/javase/specs/.

API
library

The *application program interface (API)*, also known as *library*, contains predefined classes and interfaces for developing Java programs. The API is still expanding. You can view the latest Java API documentation at https://docs.oracle.com/en/java/javase/11/.

Java is a full-fledged and powerful language that can be used in many ways. It comes in three editions:

Java SE, EE, and ME

- *Java Standard Edition (Java SE)* to develop client-side applications. The applications can run on desktop.
- *Java Enterprise Edition (Java EE)* to develop server-side applications, such as Java servlets, JavaServer Pages (JSP), and JavaServer Faces (JSF).
- *Java Micro Edition (Java ME)* to develop applications for mobile devices, such as cell phones.

This book uses Java SE to introduce Java programming. Java SE is the foundation upon which all other Java technology is based. There are many versions of Java SE. The latest, Java Development Toolkit (JDK)

Java SE 11 (or simply Java 11), is used in this book. Oracle releases each version with a *Java Development Toolkit (JDK)*. For Java 11, the Java Development Toolkit is called *JDK 11*.

Java Runtime Environment (JRE)

Integrated development environment

The JDK consists of a set of separate programs, each invoked from a command line, for compiling, running, and testing Java programs. The program for running Java programs is known as *Java Runtime Environment (JRE)*. Instead of using the JDK, you can use a Java development tool (e.g., NetBeans, Eclipse, and TextPad)—software that provides an *integrated development environment (IDE)* for developing Java programs quickly. Editing, compiling, building, debugging, and online help are integrated in one graphical user interface. You simply enter source code in one window or open an existing file in a window, and then click a button or menu item or press a function key to compile and run the program.

Check Point

1.6.1 What is the Java language specification?

1.6.2 What does JDK stand for? What does JRE stand for?

1.6.3 What does IDE stand for?

1.6.4 Are tools like NetBeans and Eclipse different languages from Java, or are they dialects or extensions of Java?

1.7 A Simple Java Program

Key Point

A Java program is executed from the **`main`** *method in the class.*

what is a console?

console input

console output

Let's begin with a simple Java program that displays the message **`Welcome to Java!`** on the console. (The word *console* is an old computer term that refers to the text entry and display device of a computer. *Console input* means to receive input from the keyboard, and *console output* means to display output on the monitor.) The program is given in Listing 1.1.

Listing 1.1 `Welcome.java`

class

`main method`

display message

VideoNote

Your first Java program

```
1  public class Welcome {
2    public static void main(String[] args) {
3      // Display message Welcome to Java! on the console
4      System.out.println("Welcome to Java!");
5    }
6  }
```

```
Welcome to Java!
```

`line numbers`

Note the *line numbers* are for reference purposes only; they are not part of the program. So, don't type line numbers in your program.

`class name`

Line 1 defines a class. Every Java program must have at least one class. Each class has a name. By convention, *class names* start with an uppercase letter. In this example, the class name is **`Welcome`**.

main method

Line 2 defines the **`main`** method. The program is executed from the **`main`** method. A class may contain several methods. The **`main`** method is the entry point where the program begins execution.

string

statement terminator

keyword

A method is a construct that contains statements. The **`main`** method in this program contains the **`System.out.println`** statement. This statement displays the string **`Welcome to Java!`** on the console (line 4). *String* is a programming term meaning a sequence of characters. A string must be enclosed in double quotation marks. Every statement in Java ends with a semicolon (**`;`**), known as the *statement terminator*.

Keywords have a specific meaning to the compiler and cannot be used for other purposes in the program. For example, when the compiler sees the word **`class`**, it understands that the word after `class` is the name for the class. Other keywords in this program are `public`, `static`, and `void`.

comment

Line 3 is a *comment* that documents what the program is and how it is constructed. Comments help programmers to communicate and understand the program. They are not programming statements, and thus are ignored by the compiler. In Java, comments are preceded by two slashes (`//`) on a line, called a *line comment*, or enclosed between `/*` and `*/` on one or several lines, called a *block comment* or *paragraph comment*. When the compiler sees `//`, it ignores all text after `//` on the same line. When it sees `/*`, it scans for the next `*/` and ignores any text between `/*` and `*/`. Here are examples of comments:

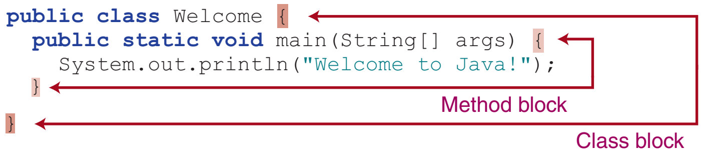
line comment

block comment

```
// This application program displays Welcome to Java!
/* This application program displays Welcome to Java! */
/* This application program
   displays Welcome to Java! */
```

block

A pair of braces in a program forms a *block* that groups the program's components. In Java, each block begins with an opening brace (`{`) and ends with a closing brace (`}`). Every class has a *class block* that groups the data and methods of the class. Similarly, every method has a *method block* that groups the statements in the method. Blocks can be *nested*, meaning that one block can be placed within another, as shown in the following code:



```
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

match braces

Tip

An opening brace must be matched by a closing brace. Whenever you type an opening brace, immediately type a closing brace to prevent the missing-brace error. Most Java IDEs automatically insert the closing brace for each opening brace.

case sensitive

Caution

Java source programs are case sensitive. It would be wrong, for example, to replace `main` in the program with `Main`.

special characters

You have seen several special characters (e.g., `{ }`, `//`, `;`) in the program. They are used in almost every program. Table 1.2 summarizes their uses.

common errors

syntax rules

The most common errors you will make as you learn to program will be syntax errors. Like any programming language, Java has its own syntax, and you need to write code that conforms to the *syntax rules*. If your program violates a rule—for example, if the semicolon is missing, a brace is missing, a quotation mark is missing, or a word is misspelled—the Java

TABLE 1.2 Special Characters

Character	*Name*	*Description*
`{}`	Opening and closing braces	Denote a block to enclose statements.
`()`	Opening and closing parentheses	Used with methods.
`[]`	Opening and closing brackets	Denote an array.
`//`	Double slashes	Precede a comment line.
`" "`	Opening and closing quotation marks	Enclose a string (i.e., sequence of characters).
`;`	Semicolon	Mark the end of a statement.

compiler will report syntax errors. Try to compile the program with these errors and see what the compiler reports.

Note

You are probably wondering why the **`main`** method is defined this way and why **`System.out.println(...)`** is used to display a message on the console. *For the time being, simply accept that this is how things are done.* Your questions will be fully answered in subsequent chapters.

The program in Listing 1.1 displays one message. Once you understand the program, it is easy to extend it to display more messages. For example, you can rewrite the program to display three messages, as shown in Listing 1.2.

LISTING 1.2 `WelcomeWithThreeMessages.java`

class
`main method`
display message

```java
public class WelcomeWithThreeMessages {
  public static void main(String[] args) {
    System.out.println("Programming is fun!");
    System.out.println("Fundamentals First");
    System.out.println("Problem Driven");
  }
}
```

```
Programming is fun!
Fundamentals First
Problem Driven
```

Further, you can perform mathematical computations and display the result on the console. Listing 1.3 gives an example of evaluating $\frac{10.5 + 2 \times 3}{45 - 3.5}$.

LISTING 1.3 `ComputeExpression.java`

class
`main method`
compute expression

```java
1  public class ComputeExpression {
2    public static void main(String[] args) {
3      System.out.print("(10.5 + 2 * 3) / (45 - 3.5) = ");
4      System.out.println((10.5 + 2 * 3) / (45 - 3.5));
5    }
6  }
```

```
(10.5 + 2 * 3) / (45 - 3.5) = 0.39759036144578314
```

`print vs. println`

The **`print`** method in line 3

```java
System.out.print("(10.5 + 2 * 3) / (45 - 3.5) = ");
```

is identical to the **`println`** method except that **`println`** moves to the beginning of the next line after displaying the string, but **`print`** does not advance to the next line when completed.

The multiplication operator in Java is **`*`**. As you can see, it is a straightforward process to translate an arithmetic expression to a Java expression. We will discuss Java expressions further in Chapter 2.

Check Point

1.7.1 What is a keyword? List some Java keywords.

1.7.2 Is Java case sensitive? What is the case for Java keywords?

1.7.3 What is a comment? Is the comment ignored by the compiler? How do you denote a comment line and a comment paragraph?

1.7.4 What is the statement to display a string on the console?

1.7.5 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    System.out.println("3.5 * 4 / 2 – 2.5 is ");
    System.out.println(3.5 * 4 / 2 – 2.5);
  }
}
```

1.8 Creating, Compiling, and Executing a Java Program

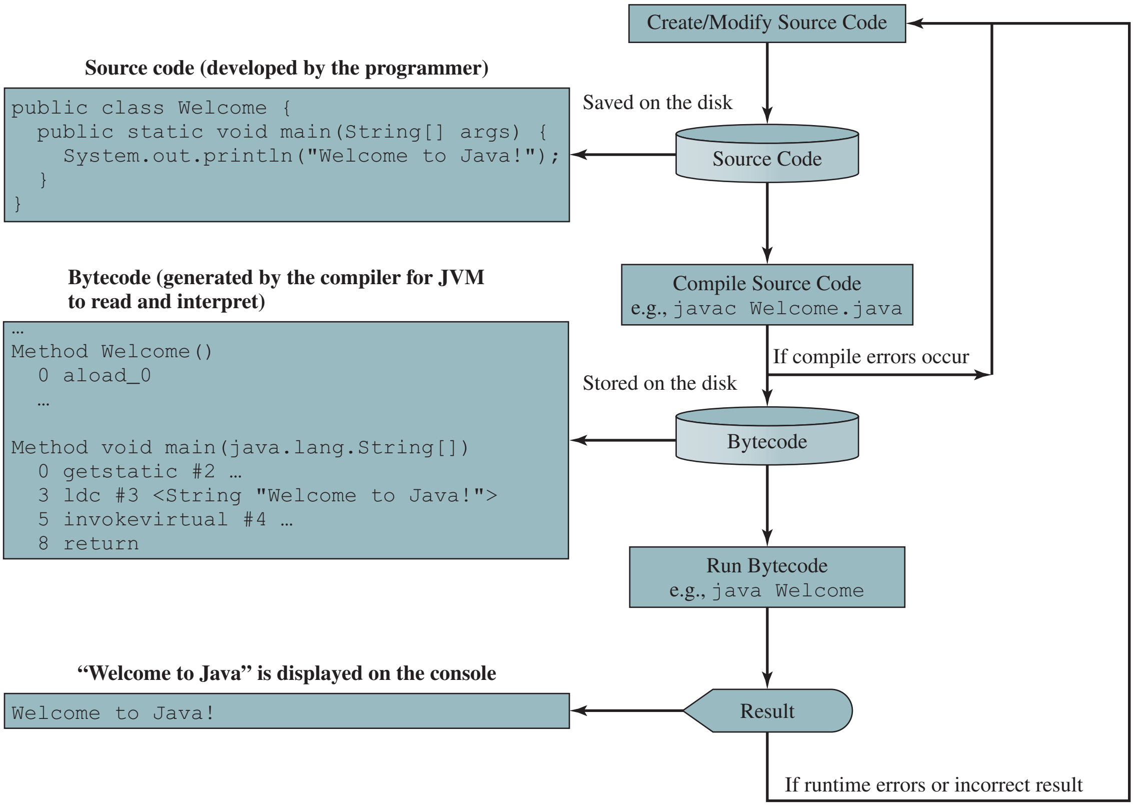
You save a Java program in a .java file and compile it into a .class file. The .class file is executed by the Java Virtual Machine (JVM).

Key Point

You have to create your program and compile it before it can be executed. This process is repetitive, as shown in Figure 1.6. If your program has compile errors, you have to modify the program to fix them, then recompile it. If your program has runtime errors or does not produce the correct result, you have to modify the program, recompile it, and execute it again.

You can use any text editor or IDE to create and edit a Java source-code file. This section demonstrates how to create, compile, and run Java programs from a command window. Sections 1.11 and 1.12 will introduce developing Java programs using NetBeans and Eclipse. From the command window, you can use a text editor such as Notepad to create the Java source-code file, as shown in Figure 1.7.

command window

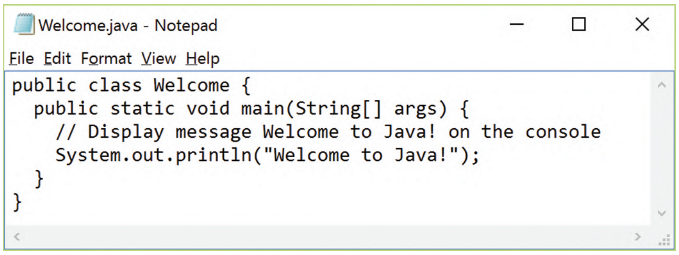

Create/Modify Source Code
Source code (developed by the programmer)
Saved on the disk
Source Code
Bytecode (generated by the compiler for JVM to read and interpret)
Compile Source Code
e.g., javac Welcome.java
If compile errors occur
Stored on the disk
Bytecode
Run Bytecode
e.g., java Welcome
"Welcome to Java" is displayed on the console
Result
If runtime errors or incorrect result

```
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

```
…
Method Welcome()
  0 aload_0
  …

Method void main(java.lang.String[])
  0 getstatic #2 …
  3 ldc #3 <String "Welcome to Java!">
  5 invokevirtual #4 …
  8 return
```

```
Welcome to Java!
```

FIGURE 1.6 The Java program-development process consists of repeatedly creating/modifying source code, compiling, and executing programs.



```
public class Welcome {
  public static void main(String[] args) {
    // Display message Welcome to Java! on the console
    System.out.println("Welcome to Java!");
  }
}
```

Figure 1.7 You can create a Java source file using Windows Notepad.

file name Welcome.java,

Note

The source file must end with the extension `.java` and must have the same exact name as the public class name. For example, the file for the source code in Listing 1.1 should be named **Welcome.java**, since the public class name is `Welcome`.

compile

A Java compiler translates a Java source file into a Java bytecode file. The following command compiles **Welcome.java**:

```
javac Welcome.java
```

Supplement I.B

Supplement I.C

Note

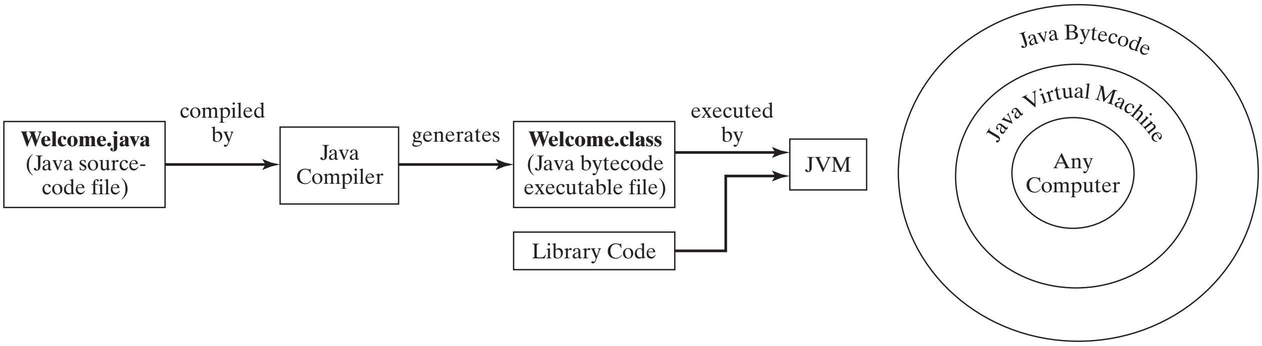
You must first install and configure the JDK before you can compile and run programs. See Supplement I.A, Installing and Configuring JDK 11, for how to install the JDK and set up the environment to compile and run Java programs. If you have trouble compiling and running programs, see Supplement I.B, Compiling and Running Java from the Command Window. This supplement also explains how to use basic DOS commands and how to use Windows Notepad to create and edit files. All the supplements are accessible from the Companion Website.

.class bytecode file

bytecode

Java Virtual Machine (JVM)

If there aren't any syntax errors, the *compiler* generates a bytecode file with a `.class` extension. Thus, the preceding command generates a file named **Welcome.class**, as shown in Figure 1.8a. The Java language is a high-level language, but Java bytecode is a low-level language. The *bytecode* is similar to machine instructions but is architecture neutral and can run on any platform that has a *Java Virtual Machine (JVM)*, as shown in Figure 1.8b. Rather than a physical machine, the virtual machine is a program that interprets Java bytecode. This is one of Java's primary advantages: *Java bytecode can run on a variety of hardware platforms and operating systems.* Java source code is compiled into Java bytecode, and Java bytecode is interpreted by the JVM. Your Java code may use the code in the Java library. The JVM executes your code along with the code in the library.



(a) (b)

Figure 1.8 (a) Java source code is translated into bytecode. (b) Java bytecode can be executed on any computer with a Java Virtual Machine.

To execute a Java program is to run the program's bytecode. You can execute the bytecode on any platform with a JVM, which is an interpreter. It translates the individual instructions in the bytecode into the target machine language code one at a time, rather than the whole program as a single unit. Each step is executed immediately after it is translated.

interpret bytecode

The following command runs the bytecode for Listing 1.1:

run

```
java Welcome
```

Figure 1.9 shows the `javac` command for compiling **Welcome.java**. The compiler generates the **Welcome.class** file, and this file is executed using the `java` command.

`javac` command
`java` command

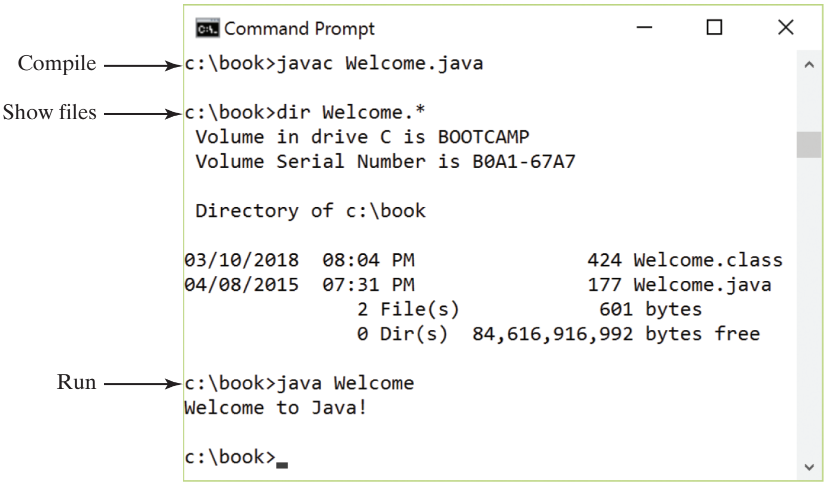
Note

For simplicity and consistency, all source-code and class files used in this book are placed under **c:\book** unless specified otherwise.

c:\book

VideoNote

Compile and Run a Java Program



```
c:\book>javac Welcome.java

c:\book>dir Welcome.*
 Volume in drive C is BOOTCAMP
 Volume Serial Number is B0A1-67A7

 Directory of c:\book

03/10/2018  08:04 PM               424 Welcome.class
04/08/2015  07:31 PM               177 Welcome.java
               2 File(s)            601 bytes
               0 Dir(s)  84,616,916,992 bytes free

c:\book>java Welcome
Welcome to Java!

c:\book>
```

FIGURE 1.9 The output of Listing 1.1 displays the message "Welcome to Java!"

Caution

Do not use the extension `.class` in the command line when executing the program. Use `java ClassName` to run the program. If you use `java ClassName.class` in the command line, the system will attempt to fetch `ClassName.class.class`.

java ClassName

Note

In JDK 11, you can use `java ClassName.java` to compile and run a single-file source code program. This command combines compiling and running in one command. A single-file source code program contains only one class in the file. This is the case for all of our programs in the first eight chapters.

Tip

If you execute a class file that does not exist, a `NoClassDefFoundError` will occur. If you execute a class file that does not have a `main` method or you mistype the `main` method (e.g., by typing `Main` instead of `main`), a `NoSuchMethodError` will occur.

NoClassDefFoundError

NoSuchMethodError

Note

When executing a Java program, the JVM first loads the bytecode of the class to memory using a program called the *class loader*. If your program uses other classes, the class loader dynamically loads them just before they are needed. After a class is loaded, the JVM uses a program called the *bytecode verifier* to check the validity of the

class loader

bytecode verifier

bytecode and to ensure that the bytecode does not violate Java's security restrictions. Java enforces strict security to make sure Java class files are not tampered with and do not harm your computer.

Pedagogical Note

use package

Your instructor may require you to use packages for organizing programs. For example, you may place all programs in this chapter in a package named *chapter 1*. For instructions on how to use packages, see Supplement I.F, Using Packages to Organize the Classes in the Text.

Check Point

1.8.1 What is the Java source filename extension, and what is the Java bytecode filename extension?

1.8.2 What are the input and output of a Java compiler?

1.8.3 What is the command to compile a Java program?

1.8.4 What is the command to run a Java program?

1.8.5 What is the JVM?

1.8.6 Can Java run on any machine? What is needed to run Java on a computer?

1.8.7 If a `NoClassDefFoundError` occurs when you run a program, what is the cause of the error?

1.8.8 If a `NoSuchMethodError` occurs when you run a program, what is the cause of the error?

1.9 Programming Style and Documentation

Key Point

Good programming style and proper documentation make a program easy to read and help programmers prevent errors.

programming style

documentation

Programming style deals with what programs look like. A program can compile and run properly even if written on only one line, but writing it all on one line would be a bad programming style because it would be hard to read. *Documentation* is the body of explanatory remarks and comments pertaining to a program. Programming style and documentation are as important as coding. Good programming style and appropriate documentation reduce the chance of errors and make programs easy to read. This section gives several guidelines. For more detailed guidelines, see Supplement I.C, Java Coding Style Guidelines, on the Companion Website.

1.9.1 Appropriate Comments and Comment Styles

Include a summary at the beginning of the program that explains what the program does, its key features, and any unique techniques it uses. In a long program, you should also include comments that introduce each major step and explain anything that is difficult to read. It is important to make comments concise so that they do not crowd the program or make it difficult to read.

javadoc comment

In addition to line comments (beginning with `//`) and block comments (beginning with `/*`), Java supports comments of a special type, referred to as *javadoc comments*. javadoc comments begin with `/**` and end with `*/`. They can be extracted into an HTML file using the JDK's `javadoc` command. For more information, see Supplement III.X, javadoc Comments, on the Companion Website.

Use javadoc comments (`/** . . . */`) for commenting on an entire class or an entire method. These comments must precede the class or the method header in order to be extracted into a javadoc HTML file. For commenting on steps inside a method, use line comments (`//`). To see an example of a javadoc HTML file, check out liveexample.pearsoncmg.com/javadoc/Exercise1.html. Its corresponding Java code is shown in liveexample.pearsoncmg.com/javadoc/Exercise1.txt.

1.9.2 Proper Indentation and Spacing

A consistent indentation style makes programs clear and easy to read, debug, and maintain. *Indentation* is used to illustrate the structural relationships between a program's components or statements. Java can read the program even if all of the statements are on the same long line, but humans find it easier to read and maintain code that is aligned properly. Indent each subcomponent or statement at least *two* spaces more than the construct within which it is nested.

indent code

A single space should be added on both sides of a binary operator, as shown in (a), rather in (b).

```
System.out.println(3 + 4 * 4);
```

(a) Good style

```
System.out.println(3+4*4);
```

(b) Bad style

1.9.3 Block Styles

A *block* is a group of statements surrounded by braces. There are two popular styles, *next-line* style and *end-of-line* style, as shown below.

```
public class Test
{
  public static void main(String[] args)
  {
    System.out.println("Block Styles");
  }
}
```

Next-line style

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Block Styles");
  }
}
```

End-of-line style

The next-line style aligns braces vertically and makes programs easy to read, whereas the end-of-line style saves space and may help avoid some subtle programming errors. Both are acceptable block styles. The choice depends on personal or organizational preference. You should use a block style consistently—mixing styles is not recommended. This book uses the *end-of-line* style to be consistent with the Java API source code.

Check Point

1.9.1 Reformat the following program according to the programming style and documentation guidelines. Use the end-of-line brace style.

```
public class Test
{
  // Main method
  public static void main(String[] args) {
  /** Display output */
  System.out.println("Welcome to Java");
  }
}
```

1.10 Programming Errors

Key Point

Programming errors can be categorized into three types: syntax errors, runtime errors, and logic errors.

1.10.1 Syntax Errors

syntax errors
compile errors

Errors that are detected by the compiler are called *syntax errors* or *compile errors*. Syntax errors result from errors in code construction, such as mistyping a keyword, omitting some necessary punctuation, or using an opening brace without a corresponding closing brace.

These errors are usually easy to detect because the compiler tells you where they are and what caused them. For example, the program in Listing 1.4 has a syntax error, as shown in Figure 1.10.

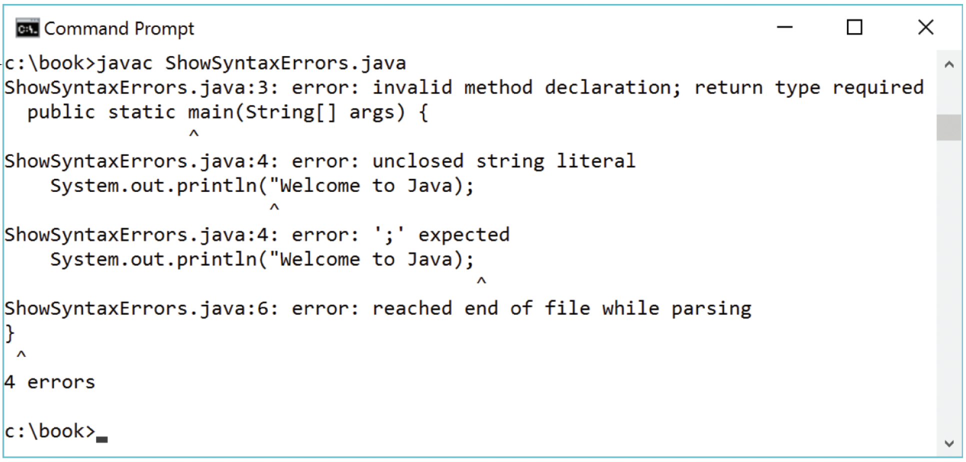
Listing 1.4 `ShowSyntaxErrors.java`

```
1  public class ShowSyntaxErrors {
2    public static main(String[] args) {
3      System.out.println("Welcome to Java);
4    }
5  }
```

Four errors are reported, but the program actually has two errors:

- The keyword `void` is missing before `main` in line 2.
- The string `Welcome to Java` should be closed with a closing quotation mark in line 3.

Since a single error will often display many lines of compile errors, it is a good practice to fix errors from the top line and work downward. Fixing errors that occur earlier in the program may also fix additional errors that occur later.



Compile

```
c:\book>javac ShowSyntaxErrors.java
ShowSyntaxErrors.java:3: error: invalid method declaration; return type required
  public static main(String[] args) {
                ^
ShowSyntaxErrors.java:4: error: unclosed string literal
    System.out.println("Welcome to Java);
                       ^
ShowSyntaxErrors.java:4: error: ';' expected
    System.out.println("Welcome to Java);
                                         ^
ShowSyntaxErrors.java:6: error: reached end of file while parsing
}
 ^
4 errors

c:\book>_
```

Figure 1.10 The compiler reports syntax errors.

fix syntax errors

Tip

If you don't know how to correct an error, compare your program closely, character by character, with similar examples in the text. In the first few weeks of this course, you will probably spend a lot of time fixing syntax errors. Soon you will be familiar with Java syntax, and can quickly fix syntax errors.

1.10.2 Runtime Errors

runtime errors

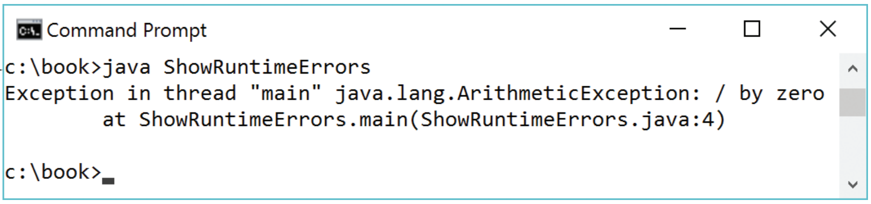
Runtime errors are errors that cause a program to terminate abnormally. They occur while a program is running if the environment detects an operation that is impossible to carry out. Input mistakes typically cause runtime errors. An *input error* occurs when the program is waiting for the user to enter a value, but the user enters a value that the program cannot handle. For instance, if the program expects to read in a number, but instead the user enters a string, this causes data-type errors to occur in the program.

Another example of runtime errors is division by zero. This happens when the divisor is zero for integer divisions. For instance, the program in Listing 1.5 would cause a runtime error, as shown in Figure 1.11.

Listing 1.5 `ShowRuntimeErrors.java`

```
public class ShowRuntimeErrors {
  public static void main(String[] args) {
    System.out.println(1 / 0);
  }
}
```

runtime error



```
c:\book>java ShowRuntimeErrors
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ShowRuntimeErrors.main(ShowRuntimeErrors.java:4)

c:\book>
```

Figure 1.11 The runtime error causes the program to terminate abnormally.

1.10.3 Logic Errors

logic errors

Logic errors occur when a program does not perform the way it was intended to. Errors of this kind occur for many different reasons. For example, suppose you wrote the program in Listing 1.6 to convert Celsius `35` degrees to a Fahrenheit degree:

Listing 1.6 `ShowLogicErrors.java`

```
public class ShowLogicErrors {
  public static void main(String[] args) {
    System.out.print("Celsius 35 is Fahrenheit degree ");
    System.out.println((9 / 5) * 35 + 32);
  }
}
```

```
Celsius 35 is Fahrenheit degree 67
```

You will get Fahrenheit `67` degrees, which is wrong. It should be `95.0`. In Java, the division for integers is the quotient—the fractional part is truncated—so in Java `9 / 5` is `1`. To get the correct result, you need to use `9.0 / 5`, which results in `1.8`.

In general, syntax errors are easy to find and easy to correct because the compiler gives indications as to where the errors came from and why they are wrong. Runtime errors are not difficult to find, either, since the reasons and locations for the errors are displayed on the console when the program aborts. Finding logic errors, on the other hand, can be very challenging. In the upcoming chapters, you will learn the techniques of tracing programs and finding logic errors.

1.10.4 Common Errors

Missing a closing brace, missing a semicolon, missing quotation marks for strings, and misspelling names are common errors for new programmers.

Common Error 1: Missing Braces

The braces are used to denote a block in the program. Each opening brace must be matched by a closing brace. A common error is missing the closing brace. To avoid this error, type a closing brace whenever an opening brace is typed, as shown in the following example:

```
public class Welcome {

}
```

Type this closing brace right away to match the opening brace

If you use an IDE such as NetBeans and Eclipse, the IDE automatically inserts a closing brace for each opening brace typed.

Common Error 2: Missing Semicolons

Each statement ends with a statement terminator (;). Often, a new programmer forgets to place a statement terminator for the last statement in a block, as shown in the following example:

```
public static void main(String[] args) {
  System.out.println("Programming is fun!");
  System.out.println("Fundamentals First");
  System.out.println("Problem Driven")
}
```

Missing a semicolon

Common Error 3: Missing Quotation Marks

A string must be placed inside the quotation marks. Often, a new programmer forgets to place a quotation mark at the end of a string, as shown in the following example:

```
System.out.println("Problem Driven );
```

Missing a quotation mark

If you use an IDE such as NetBeans and Eclipse, the IDE automatically inserts a closing quotation mark for each opening quotation mark typed.

Common Error 4: Misspelling Names

Java is case sensitive. Misspelling names is a common error for new programmers. For example, the word `main` is misspelled as `Main` and `String` is misspelled as `string` in the following code:

```
public class Test {
  public static void Main(string[] args) {
    System.out.println((10.5 + 2 * 3) / (45 - 3.5));
  }
}
```

Check Point

1.10.1 What are syntax errors (compile errors), runtime errors, and logic errors?

1.10.2 Give examples of syntax errors, runtime errors, and logic errors.

1.10.3 If you forget to put a closing quotation mark on a string, what kind of error will be raised?

1.10.4 If your program needs to read integers, but the user entered strings, an error would occur when running this program. What kind of error is this?

1.10.5 Suppose you write a program for computing the perimeter of a rectangle and you mistakenly write your program so it computes the area of a rectangle. What kind of error is this?

1.10.6 Identify and fix the errors in the following code:

```
public class Welcome {
  public void Main(String[] args) {
    System.out.println('Welcome to Java!);
  }
)
```

1.11 Developing Java Programs Using NetBeans

You can edit, compile, run, and debug Java Programs using NetBeans.

Note

Section 1.8 introduced developing programs from the command line. Many of our readers also use an IDE. This section and next section introduce two most popular Java IDEs: NetBeans and Eclipse. These two sections may be skipped.

Key Point

NetBeans and Eclipse are two free popular integrated development environments for developing Java programs. They are easy to learn if you follow simple instructions. We recommend that you use either one for developing Java programs. This section gives the essential instructions to guide new users to create a project, create a class, compile, and run a class in NetBeans. The use of Eclipse will be introduced in the next section. To use JDK 11, you need NetBeans 9 or higher. For instructions on downloading and installing latest version of NetBeans, see Supplement II.B.

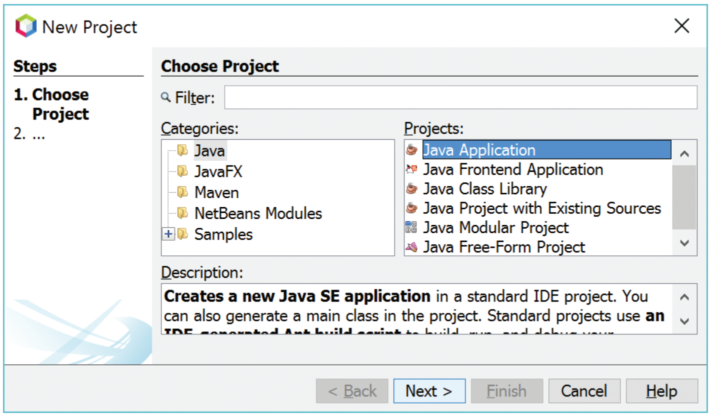
VideoNote

NetBeans brief tutorial

1.11.1 Creating a Java Project

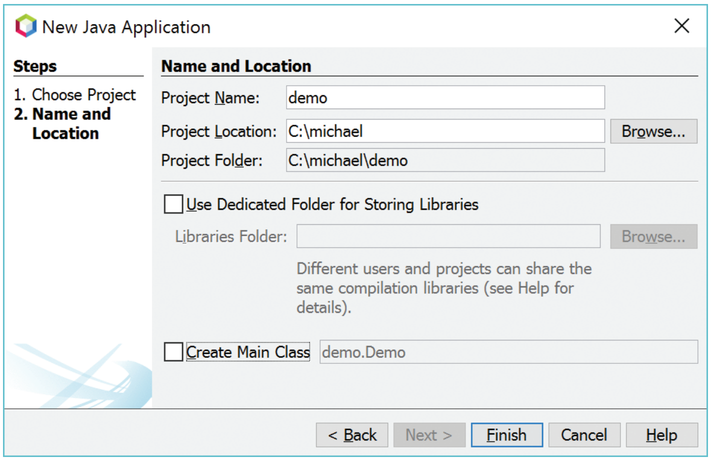
Before you can create Java programs, you need to first create a project. A project is like a folder to hold Java programs and all supporting files. You need to create a project only once. Here are the steps to create a Java project:

1. Choose *File*, *New Project* to display the New Project dialog box, as shown in Figure 1.12.
2. Select Java in the Categories section and Java Application in the Projects section, and then click *Next* to display the New Java Application dialog box, as shown in Figure 1.13.
3. Type `demo` in the Project Name field and `c:\michael` in Project Location field. Uncheck *Use Dedicated Folder for Storing Libraries* and uncheck *Create Main Class*.
4. Click *Finish* to create the project, as shown in Figure 1.14.


New Project
Steps
1. Choose Project
2. ...
Choose Project
Filter:
Categories:
Java
JavaFX
Maven
NetBeans Modules
Samples
Projects:
Java Application
Java Frontend Application
Java Class Library
Java Project with Existing Sources
Java Modular Project
Java Free-Form Project
Description:
< Back
Next >
Finish
Cancel
Help

Creates a new Java SE application in a standard IDE project. You can also generate a main class in the project. Standard projects use **an**

Figure 1.12 The New Project dialog is used to create a new project and specify a project type. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.



New Java Application

Steps
1. Choose Project
2. Name and Location

Name and Location

Project Name: demo

Project Location: C:\michael

Project Folder: C:\michael\demo

☐ Use Dedicated Folder for Storing Libraries

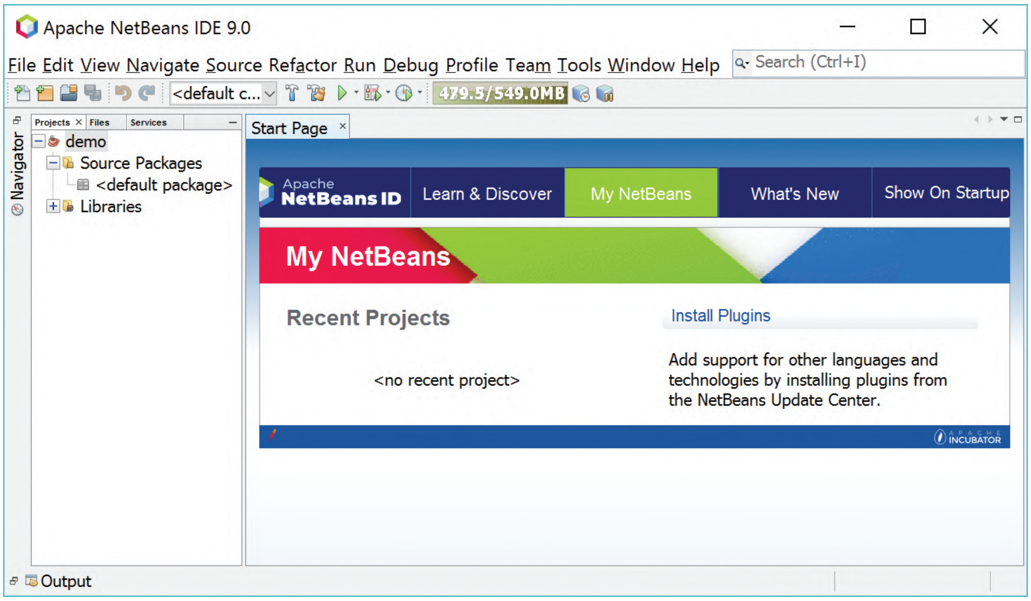
Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

☐ Create Main Class demo.Demo

Browse...
Browse...
< Back
Next >
Finish
Cancel
Help

Figure 1.13 The New Java Application dialog is for specifying a project name and location. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.



Apache NetBeans IDE 9.0

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default c...
479.5/549.0MB
Navigator
Projects
Files
Services
demo
Source Packages
<default package>
Libraries
Start Page
Apache NetBeans ID
Learn & Discover
My NetBeans
What's New
Show On Startup
My NetBeans
Recent Projects
<no recent project>
Install Plugins
Add support for other languages and technologies by installing plugins from the NetBeans Update Center.
Apache Incubator
Output

Figure 1.14 A New Java project named demo is created. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

1.11.2 Creating a Java Class

After a project is created, you can create Java programs in the project using the following steps:

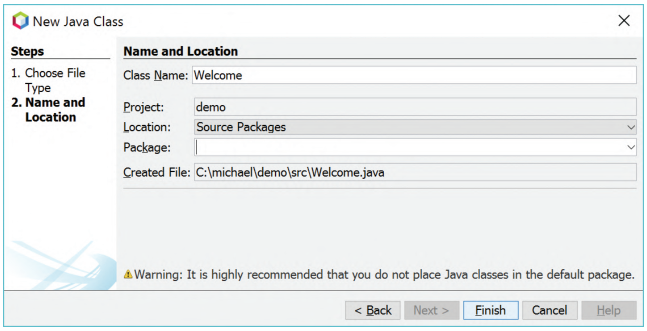
1. Right-click the demo node in the project pane to display a context menu. Choose *New*, *Java Class* to display the New Java Class dialog box, as shown in Figure 1.15.
2. Type `Welcome` in the Class Name field and select the Source Packages in the Location field. Leave the Package field blank. This will create a class in the default package.

3. Click *Finish* to create the Welcome class. The source-code file **Welcome.java** is placed under the <default package> node.

4. Modify the code in the Welcome class to match Listing 1.1 in the text, as shown in Figure 1.16.

1.11.3 Compiling and Running a Class

To run **Welcome.java**, right-click **Welcome.java** to display a context menu and choose *Run File*, or simply press Shift + F6. The output is displayed in the Output pane, as shown in Figure 1.16. The *Run File* command automatically compiles the program if the program has been changed.


New Java Class
Steps
1. Choose File Type
2. Name and Location
Name and Location
< Back
Next >
Finish
Cancel
Help

Class Name: Welcome

Project: demo

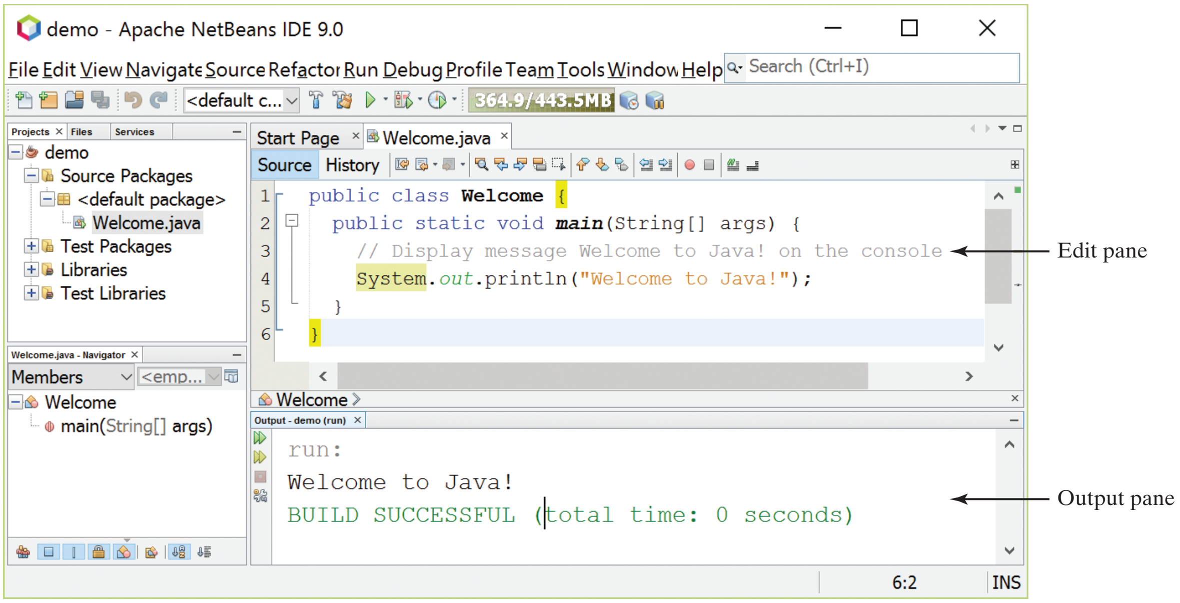
Location: Source Packages

Package:

Created File: C:\michael\demo\src\Welcome.java

⚠Warning: It is highly recommended that you do not place Java classes in the default package.

FIGURE 1.15 The New Java Class dialog box is used to create a new Java class. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.


demo - Apache NetBeans IDE 9.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default c...
364.9/443.5MB
Projects
Files
Services
demo
Source Packages
<default package>
Welcome.java
Test Packages
Libraries
Test Libraries
Welcome.java - Navigator
Members
<emp...
Welcome
main(String[] args)
Start Page
Welcome.java
Source
History
Welcome
Output - demo (run)
6:2
INS
Edit pane
Output pane

```
public class Welcome {
  public static void main(String[] args) {
    // Display message Welcome to Java! on the console
    System.out.println("Welcome to Java!");
  }
}
```

```
run:
Welcome to Java!
BUILD SUCCESSFUL (total time: 0 seconds)
```

FIGURE 1.16 You can edit a program and run it in NetBeans. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

1.12 Developing Java Programs Using Eclipse

You can edit, compile, run, and debug Java Programs using Eclipse.

Key Point

The preceding section introduced developing Java programs using NetBeans. You can also use Eclipse to develop Java programs. This section gives the essential instructions to guide new users to create a project, create a class, and compile/run a class in Eclipse. To use JDK 11, you need Eclipse 4.9 or higher. For instructions on downloading and installing latest version of Eclipse, see Supplement II.D.

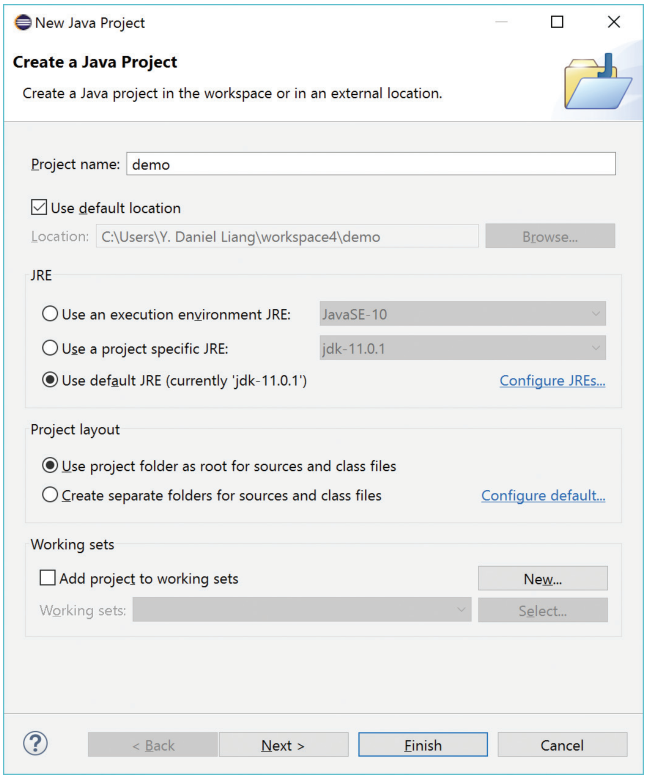
VideoNote

Eclipse brief tutorial

1.12.1 Creating a Java Project

Before creating Java programs in Eclipse, you need to first create a project to hold all files. Here are the steps to create a Java project in Eclipse:

1. Choose *File*, *New*, *Java Project* to display the New Project wizard, as shown in Figure 1.17.



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: demo

☑ Use default location

Location: C:\Users\Y. Daniel Liang\workspace4\demo

JRE

○ Use an execution environment JRE: JavaSE-10

○ Use a project specific JRE: jdk-11.0.1

● Use default JRE (currently 'jdk-11.0.1')

Project layout

● Use project folder as root for sources and class files

○ Create separate folders for sources and class files

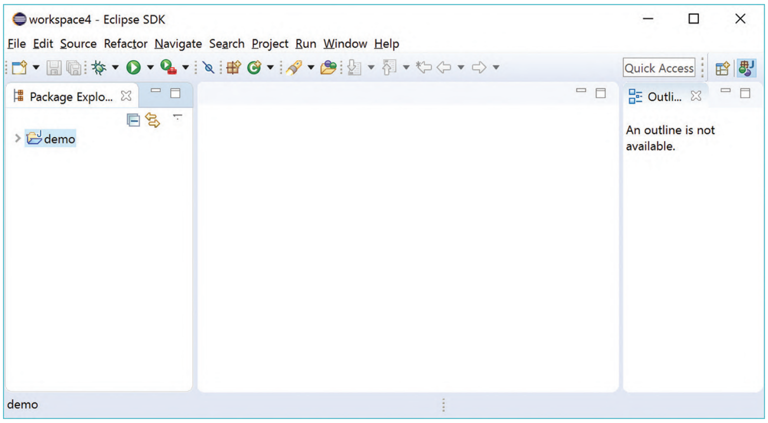
Working sets

☐ Add project to working sets

Working sets:

FIGURE 1.17 The New Java Project dialog is for specifying a project name and the properties. *Source*: Eclipse Foundation, Inc.

2. Type **`demo`** in the Project name field. As you type, the Location field is automatically set by default. You may customize the location for your project.
3. Make sure you selected the options *Use project folder as root for sources and class files* so the .java and .class files are in the same folder for easy access.
4. Click *Finish* to create the project, as shown in Figure 1.18.


workspace4 - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explo...
demo
Outli...
demo

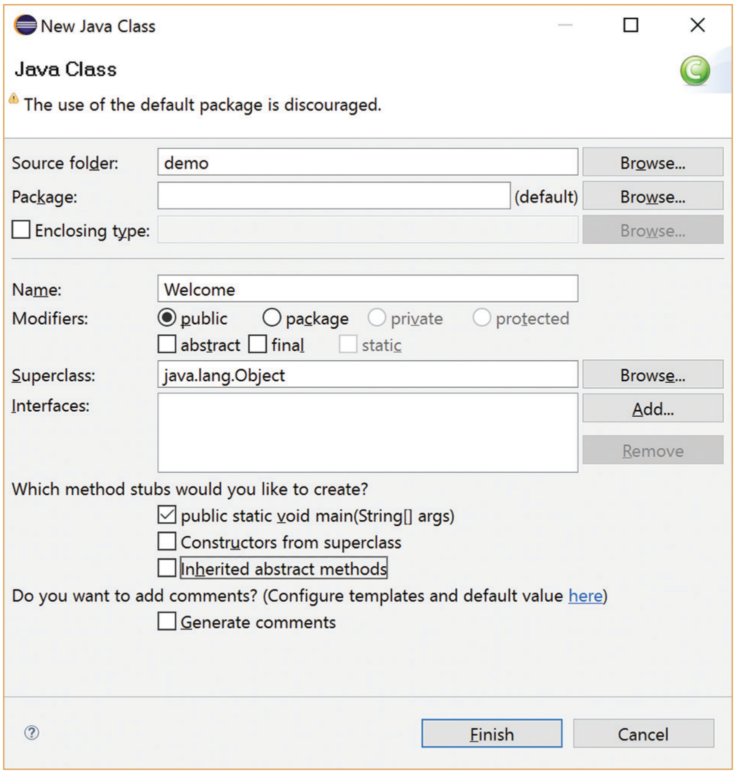
An outline is not available.

Figure 1.18 A New Java project named demo is created. *Source*: Eclipse Foundation, Inc.

1.12.2 Creating a Java Class

After a project is created, you can create Java programs in the project using the following steps:

1. Choose *File*, *New*, *Class* to display the New Java Class wizard.
2. Type `Welcome` in the Name field.
3. Check the option *public static void main(String[] args)*.
4. Click *Finish* to generate the template for the source code **Welcome.java**, as shown in Figure 1.19.


New Java Class
Java Class
Browse...
Browse...
Browse...
Browse...
Add...
Remove
Finish
Cancel

The use of the default package is discouraged.

Source folder: demo

Package: (default)

☐ Enclosing type:

Name: Welcome

Modifiers: ● public ○ package ○ private ○ protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

☐ Constructors from superclass

☐ Inherited abstract methods

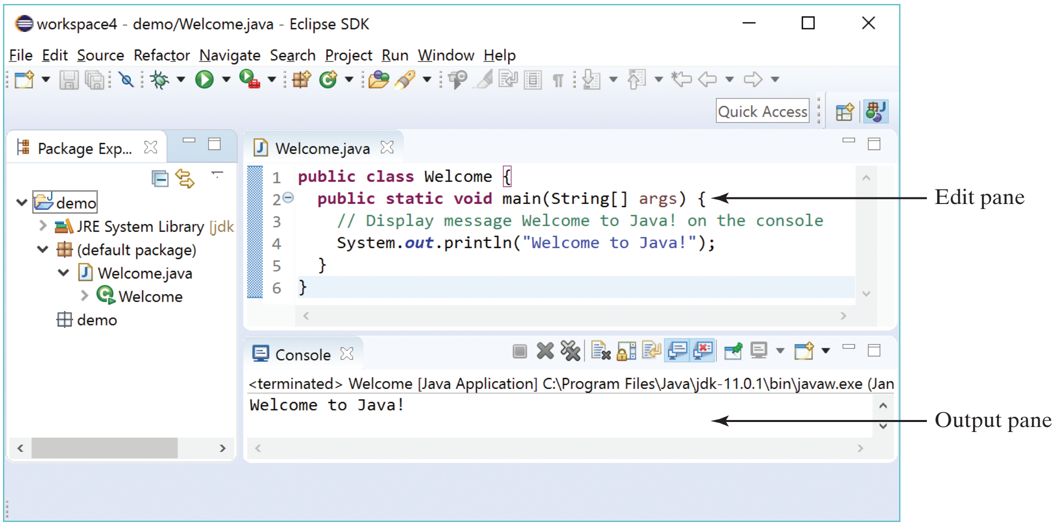
Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Figure 1.19 The New Java Class dialog box is used to create a new Java class. *Source*: Eclipse Foundation, Inc.

1.12.3 Compiling and Running a Class

To run the program, right-click the class in the project to display a context menu. Choose *Run*, *Java Application* in the context menu to run the class. The output is displayed in the Console pane, as shown in Figure 1.20. The *Run* command automatically compiles the program if the program has been changed.


workspace4 - demo/Welcome.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Exp...
demo
JRE System Library [jdk
(default package)
Welcome.java
Welcome
demo
Welcome.java
Console
Edit pane
Output pane

```
public class Welcome {
  public static void main(String[] args) {
    // Display message Welcome to Java! on the console
    System.out.println("Welcome to Java!");
  }
}
```

```
<terminated> Welcome [Java Application] C:\Program Files\Java\jdk-11.0.1\bin\javaw.exe (Jan
Welcome to Java!
```

FIGURE 1.20 You can edit a program and run it in Eclipse. *Source*: Eclipse Foundation, Inc.

KEY TERMS

Application Program Interface (API)	11
assembler	7
assembly language	7
bit	3
block	13
block comment	13
bus	2
byte	3
bytecode	16
bytecode verifier	17
cable modem	6
central processing unit (CPU)	3
class loader	17
comment	13
compiler	8
console	12
dial-up modem	6
dot pitch	6
DSL (digital subscriber line)	6
encoding scheme	3
hardware	2
high-level language	8
integrated development environment (IDE)	12
interpreter	8
java *command*	17
Java Development Toolkit (JDK)	12
Java language specification	11
Java Runtime Environment (JRE)	12
Java Virtual Machine (JVM)	16
javac *command*	17
keyword	12
library	11
line comment	13
logic error	21
low-level language	8
machine language	7
main *method*	12
memory	4
motherboard	3
network interface card (NIC)	6
operating system (OS)	9
pixel	6
program	2
programming	2
runtime error	20
screen resolution	6
software	2

source code	8
source program	8
statement	8
statement terminator	12
storage devices	4
syntax error	19

Note

The above terms are defined in this chapter. Glossary (at the end of TOC) lists all the key terms and descriptions in the book, organized by chapters.

Supplement I.A

CHAPTER SUMMARY

1. A computer is an electronic device that stores and processes data.

2. A computer includes both *hardware* and *software*.

3. Hardware is the physical aspect of the computer that can be touched.

4. Computer *programs*, known as *software*, are the invisible instructions that control the hardware and make it perform tasks.

5. Computer *programming* is the writing of instructions (i.e., code) for computers to perform.

6. The *central processing unit (CPU)* is a computer's brain. It retrieves instructions from *memory* and executes them.

7. Computers use zeros and ones because digital devices have two stable states, referred to by convention as zero and one.

8. A *bit* is a binary digit 0 or 1.

9. A *byte* is a sequence of 8 bits.

10. A kilobyte is about 1,000 bytes, a megabyte about 1 million bytes, a gigabyte about 1 billion bytes, and a terabyte about 1,000 gigabytes.

11. Memory stores data and program instructions for the CPU to execute.

12. A memory unit is an ordered sequence of bytes.

13. Memory is volatile, because information is lost when the power is turned off.

14. Programs and data are permanently stored on *storage devices* and are moved to memory when the computer actually uses them.

15. The *machine language* is a set of primitive instructions built into every computer.

16. *Assembly language* is a *low-level programming language* in which a mnemonic is used to represent each machine-language instruction.

17. *High-level languages* are English-like and easy to learn and program.

18. A program written in a high-level language is called a *source program.*

19. A *compiler* is a software program that translates the source program into a *machine-language program.*

20. The *operating system (OS)* is a program that manages and controls a computer's activities.

21. Java is platform independent, meaning you can write a program once and run it on any computer.
22. The Java source file name must match the public class name in the program. Java source-code files must end with the `.java` extension.
23. Every class is compiled into a separate bytecode file that has the same name as the class and ends with the `.class` extension.
24. To compile a Java source-code file from the command line, use the `javac` command.
25. To run a Java class from the command line, use the `java` command.
26. Every Java program is a set of class definitions. The keyword `class` introduces a class definition. The contents of the class are included in a *block*.
27. A block begins with an opening brace (`{`) and ends with a closing brace (`}`).
28. Methods are contained in a class. To run a Java program, the program must have a `main` method. The `main` method is the entry point where the program starts when it is executed.
29. Every *statement* in Java ends with a semicolon (`;`), known as the *statement terminator.*
30. *Keywords* have a specific meaning to the compiler and cannot be used for other purposes in the program.
31. In Java, comments are preceded by two slashes (`//`) on a line, called a *line comment,* or enclosed between `/*` and `*/` on one or several lines, called a *block comment* or *paragraph comment*. Comments are ignored by the compiler.
32. Java source programs are case sensitive.
33. Programming errors can be categorized into three types: *syntax errors*, *runtime errors*, and *logic errors*. Errors reported by a compiler are called syntax errors or *compile errors*. Runtime errors are errors that cause a program to terminate abnormally. Logic errors occur when a program does not perform the way it was intended to.

QUIZ

Answer the quiz for this chapter at www.pearsonhighered.com/liang. Choose this book and click Companion Website to select Quiz.

MyProgrammingLab™

PROGRAMMING EXERCISES

Pedagogical Note

We cannot stress enough the importance of learning programming through exercises. For this reason, the book provides a large number of programming exercises at various levels of difficulty. The problems cover many application areas, including math, science, business, financial, gaming, animation, and multimedia. Solutions to most even-numbered programming exercises are on the Companion Website. Solutions to most odd-numbered programming exercises are on the Instructor Resource Website. The level of difficulty is rated easy (no star), moderate (*****), hard (******), or challenging (*******).

level of difficulty

1.1 (*Display three messages*) Write a program that displays `Welcome to Java`, `Welcome to Computer Science`, and `Programming is fun`.

1.2 (*Display five messages*) Write a program that displays `Welcome to Java` five times.

***1.3** (*Display a pattern*) Write a program that displays the following pattern:

```
    J     A     V     V     A
    J    A A     V   V     A A
J   J   AAAAA     V V     AAAAA
 J J   A     A     V     A     A
```

1.4 (*Print a table*) Write a program that displays the following table:

```
a      a^2    a^3
1      1      1
2      4      8
3      9      27
4      16     64
```

1.5 (*Compute expressions*) Write a program that displays the result of

$$\frac{9.5 \times 4.5 - 2.5 \times 3}{45.5 - 3.5}.$$

1.6 (*Summation of a series*) Write a program that displays the result of

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9.$$

1.7 (*Approximate* π) π can be computed using the following formula:

$$\pi = 4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \ldots\right)$$

Write a program that displays the result of $4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11}\right)$ and $4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13}\right)$. Use `1.0` instead of `1` in your program.

1.8 (*Area and perimeter of a circle*) Write a program that displays the area and perimeter of a circle that has a radius of `5.5` using the following formulas:

$$perimeter = 2 \times radius \times \pi$$

$$area = radius \times radius \times \pi$$

1.9 (*Area and perimeter of a rectangle*) Write a program that displays the area and perimeter of a rectangle with a width of `4.5` and a height of `7.9` using the following formula:

$$area = width \times height$$

1.10 (*Average speed in miles*) Assume that a runner runs `14` kilometers in `45` minutes and `30` seconds. Write a program that displays the average speed in miles per hour. (Note `1` mile is equal to `1.6` kilometers.)

***1.11** (*Population projection*) The U.S. Census Bureau projects population based on the following assumptions:

- One birth every 7 seconds
- One death every 13 seconds
- One new immigrant every 45 seconds

Write a program to display the population for each of the next five years. Assume that the current population is 312,032,486, and one year has 365 days. *Hint*: In Java, if two integers perform division, the result is an integer. The fractional part is truncated. For example, `5 / 4` is `1` (not `1.25`) and `10 / 4` is `2` (not `2.5`). To get an accurate result with the fractional part, one of the values involved in the division must be a number with a decimal point. For example, `5.0 / 4` is `1.25` and `10 / 4.0` is `2.5`.

1.12 (*Average speed in kilometers*) Assume that a runner runs `24` miles in `1` hour, `40` minutes, and `35` seconds. Write a program that displays the average speed in kilometers per hour. (Note `1` mile is equal to `1.6` kilometers.)

***1.13** (*Algebra: solve* 2 × 2 *linear equations*) You can use Cramer's rule to solve the following 2 × 2 system of linear equation provided that $ad - bc$ is not 0:

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \qquad x = \frac{ed - bf}{ad - bc} \qquad y = \frac{af - ec}{ad - bc}$$

Write a program that solves the following equation and displays the value for x and y: (Hint: replace the symbols in the formula with numbers to compute x and y. This exercise can be done in Chapter 1 without using materials in later chapters.)

$$\begin{aligned} 3.4x + 50.2y &= 44.5 \\ 2.1x + .55y &= 5.9 \end{aligned}$$

Note

More than 200 additional programming exercises with solutions are provided to the instructors on the Instructor Resource Website.

CHAPTER

2

ELEMENTARY PROGRAMMING

Objectives

- To write Java programs to perform simple computations (§2.2).
- To obtain input from the console using the `Scanner` class (§2.3).
- To use identifiers to name variables, constants, methods, and classes (§2.4).
- To use variables to store data (§§2.5 and 2.6).
- To program with assignment statements and assignment expressions (§2.6).
- To use constants to store permanent data (§2.7).
- To name classes, methods, variables, and constants by following their naming conventions (§2.8).
- To explore Java numeric primitive data types: `byte`, `short`, `int`, `long`, `float`, and `double` (§2.9).
- To read a `byte`, `short`, `int`, `long`, `float`, or `double` value from the keyboard (§2.9.1).
- To perform operations using operators `+`, `−`, `*`, `/`, and `%` (§2.9.2).
- To perform exponent operations using `Math.pow(a, b)` (§2.9.3).
- To write integer literals, floating-point literals, and literals in scientific notation (§2.10).
- To use JShell to quickly test Java code (§2.11).
- To write and evaluate numeric expressions (§2.12).
- To obtain the current system time using `System.currentTimeMillis()` (§2.13).
- To use augmented assignment operators (§2.14).
- To distinguish between postincrement and preincrement and between postdecrement and predecrement (§2.15).
- To cast the value of one type to another type (§2.16).
- To describe the software development process and apply it to develop the loan payment program (§2.17).
- To write a program that converts a large amount of money into smaller units (§2.18).
- To avoid common errors and pitfalls in elementary programming (§2.19).

2.1 Introduction

Key Point

The focus of this chapter is on learning elementary programming techniques to solve problems.

In Chapter 1, you learned how to create, compile, and run very basic Java programs. You will learn how to solve problems by writing programs. Through these problems, you will learn elementary programming using primitive data types, variables, constants, operators, expressions, and input and output.

Suppose, for example, you need to take out a student loan. Given the loan amount, loan term, and annual interest rate, can you write a program to compute the monthly payment and total payment? This chapter shows you how to write programs like this. Along the way, you will learn the basic steps that go into analyzing a problem, designing a solution, and implementing the solution by creating a program.

2.2 Writing a Simple Program

Key Point

Writing a program involves designing a strategy for solving the problem then using a programming language to implement that strategy.

problem

Let's first consider the simple *problem* of computing the area of a circle. How do we write a program for solving this problem?

algorithm

pseudocode

Writing a program involves designing algorithms and translating algorithms into programming instructions, or code. An *algorithm* lists the steps you can follow to solve a problem. Algorithms can help the programmer plan a program before writing it in a programming language. Algorithms can be described in natural languages or in *pseudocode* (natural language mixed with some programming code). The algorithm for calculating the area of a circle can be described as follows:

1. Read in the circle's radius.
2. Compute the area using the following formula:

$$area = radius \times radius \times \pi$$

3. Display the result.

Tip

It's always a good practice to outline your program (or its underlying problem) in the form of an algorithm before you begin coding.

When you *code*—that is, when you write a program—you translate an algorithm into a program. You already know every Java program begins with a class definition in which the keyword `class` is followed by the class name. Assume you have chosen `ComputeArea` as the class name. The outline of the program would look as follows:

```
public class ComputeArea {
  // Details to be given later
}
```

As you know, every Java program must have a `main` method where program execution begins. The program is then expanded as follows:

```
public class ComputeArea {
  public static void main(String[] args) {
    // Step 1: Read in radius

    // Step 2: Compute area
    // Step 3: Display the area
  }
}
```

The program needs to read the radius entered by the user from the keyboard. This raises two important issues:

- Reading the radius
- Storing the radius in the program

Let's address the second issue first. In order to store the radius, the program needs to declare a symbol called a *variable*. A variable represents a value stored in the computer's memory.

variable

Rather than using **`x`** and **`y`** as variable names, choose *descriptive names*: in this case, **`radius`** for radius and **`area`** for area. To let the compiler know what **`radius`** and **`area`** are, specify their *data types*. That is the kind of data stored in a variable, whether an integer, real number, or something else. This is known as *declaring variables*. Java provides simple data types for representing integers, real numbers, characters, and Boolean types. These types are known as *primitive data types* or *fundamental types*.

descriptive names

data type

declare variables

primitive data types

Real numbers (i.e., numbers with a decimal point) are represented using a method known as *floating-point* in computers. Therefore, the real numbers are also called *floating-point numbers*. In Java, you can use the keyword **`double`** to declare a floating-point variable. Declare **`radius`** and **`area`** as **`double`**. The program can be expanded as follows:

floating-point numbers

```
public class ComputeArea {
  public static void main(String[] args) {
    double radius;
    double area;

    // Step 1: Read in radius

    // Step 2: Compute area

    // Step 3: Display the area
  }
}
```

The program declares **`radius`** and **`area`** as variables. The keyword **`double`** indicates that **`radius`** and **`area`** are floating-point values stored in the computer.

The first step is to prompt the user to designate the circle's **`radius`**. You will soon learn how to prompt the user for information. For now, to learn how variables work, you can assign a fixed value to **`radius`** in the program as you write the code. Later, you'll modify the program to prompt the user for this value.

The second step is to compute **`area`** by assigning the result of the expression **`radius * radius * 3.14159`** to **`area`**.

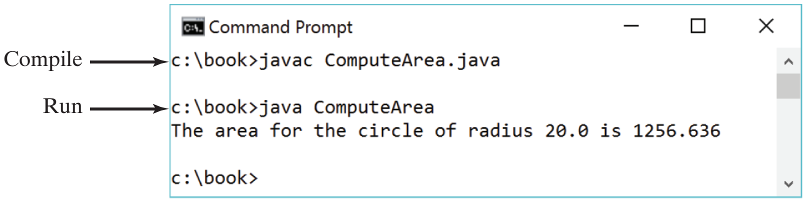
In the final step, the program will display the value of **`area`** on the console by using the **`System.out.println`** method.

Listing 2.1 shows the complete program, and a sample run of the program is shown in Figure 2.1.

LISTING 2.1 ComputeArea.java

```
public class ComputeArea {
  public static void main(String[] args) {
    double radius; // Declare radius
    double area; // Declare area

    // Assign a radius
    radius = 20; // radius is now 20
```

```
 8
 9      // Compute area
10      area = radius * radius * 3.14159;
11
12      // Display results
13      System.out.println("The area for the circle of radius " +
14        radius + " is " + area);
15    }
16  }
```



```
c:\book>javac ComputeArea.java

c:\book>java ComputeArea
The area for the circle of radius 20.0 is 1256.636

c:\book>
```

FIGURE 2.1 The program displays the area of a circle.

declare variable
assign value

Variables such as `radius` and `area` correspond to memory locations. Every variable has a name, a type, and a value. Line 3 declares that `radius` can store a `double` value. The value is not defined until you assign a value. Line 7 assigns `20` into the variable `radius`. Similarly, line 4 declares the variable `area`, and line 10 assigns a value into `area`. The following table shows the value in the memory for `area` and `radius` as the program is executed. Each row in the table shows the values of variables after the statement in the corresponding line in the program is executed. This method of reviewing how a program works is called *tracing a program*. Tracing programs are helpful for understanding how programs work, and they are useful tools for finding errors in programs.

tracing program

line#	radius	area
3	no value	
4		no value
7	20	
10		1256.636

concatenate strings

concatenate strings with numbers

The plus sign (`+`) has two meanings: one for addition, and the other for concatenating (combining) strings. The plus sign (`+`) in lines 13–14 is called a *string concatenation operator*. It combines two strings into one. If a string is combined with a number, the number is converted into a string and concatenated with the other string. Therefore, the plus signs (`+`) in lines 13–14 concatenate strings into a longer string, which is then displayed in the output. Strings and string concatenation will be discussed further in Chapter 4.

Caution

A string cannot cross lines in the source code. Thus, the following statement would result in a compile error:

```
System.out.println("Introduction to Java Programming,
by Y. Daniel Liang");
```

break a long string

To fix the error, break the string into separate substrings, and use the concatenation operator (`+`) to combine them:

```
System.out.println("Introduction to Java Programming, " +
  "by Y. Daniel Liang");
```

Check Point

2.2.1 Identify and fix the errors in the following code:

```
public class Test {
  public void main(string[] args) {
    double i = 50.0;
    double k = i + 50.0;
    double j = k + 1;

    System.out.println("j is " + j + " and
      k is " + k);
  }
}
```

2.3 Reading Input from the Console

Reading input from the console enables the program to accept input from the user.

Key Point

VideoNote
Obtain Input

In Listing 2.1, the radius is fixed in the source code. To use a different radius, you have to modify the source code and recompile it. Obviously, this is not convenient, so instead you can use the `Scanner` class for console input.

Java uses `System.out` to refer to the standard output device, and `System.in` to the standard input device. By default, the output device is the display monitor, and the input device is the keyboard. To perform console output, you simply use the `println` method to display a primitive value or a string to the console. To perform console input, you need to use the `Scanner` class to create an object to read input from `System.in`, as follows:

```
Scanner input = new Scanner(System.in);
```

The syntax `new Scanner(System.in)` creates an object of the `Scanner` type. The syntax `Scanner input` declares that `input` is a variable whose type is `Scanner`. The whole line `Scanner input = new Scanner(System.in)` creates a `Scanner` object and assigns its reference to the variable `input`. An object may invoke its methods. To invoke a method on an object is to ask the object to perform a task. You can invoke the `nextDouble()` method to read a `double` value as follows:

```
double radius = input.nextDouble();
```

This statement reads a number from the keyboard and assigns the number to `radius`.

Listing 2.2 rewrites Listing 2.1 to prompt the user to enter a radius.

LISTING 2.2 ComputeAreaWithConsoleInput.java

import class

create a `Scanner`

read a `double`

```
import java.util.Scanner; // Scanner is in the java.util package

public class ComputeAreaWithConsoleInput {
  public static void main(String[] args) {
    // Create a Scanner object
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a radius
    System.out.print("Enter a number for radius: ");
    double radius = input.nextDouble();

    // Compute area
    double area = radius * radius * 3.14159;

    // Display results
    System.out.println("The area for the circle of radius " +
17         radius + " is " + area);
18     }
19   }
```

```
Enter a number for radius: 2.5 ↵ Enter
The area for the circle of radius 2.5 is 19.6349375
```

```
Enter a number for radius: 23 ↵ Enter
The area for the circle of radius 23.0 is 1661.90111
```

The `Scanner` class is in the `java.util` package. It is imported in line 1. Line 6 creates a `Scanner` object. Note the `import` statement can be omitted if you replace `Scanner` by `java.util.Scanner` in line 6.

prompt

Line 9 displays a string `"Enter a number for radius: "` to the console. This is known as a *prompt*, because it directs the user to enter an input. Your program should always tell the user what to enter when expecting input from the keyboard.

Recall that the `print` method in line 9 is identical to the `println` method, except that `println` moves to the beginning of the next line after displaying the string, but `print` does not advance to the next line when completed.

Line 6 creates a `Scanner` object. The statement in line 10 reads input from the keyboard.

```
double radius = input.nextDouble();
```

After the user enters a number and presses the *Enter* key, the program reads the number and assigns it to `radius`.

More details on objects will be introduced in Chapter 9. *For the time being, simply accept that this is how we obtain input from the console*.

specific import

The `Scanner` class is in the `java.util` package. It is imported in line 1. There are two types of `import` statements: *specific import* and *wildcard import.* The *specific import* specifies a single class in the import statement. For example, the following statement imports `Scanner` from the package `java.util`.

```
import java.util.Scanner;
```

wildcard import

The *wildcard import* imports all the classes in a package by using the asterisk as the wildcard. For example, the following statement imports all the classes from the package `java.util`.

```
import java.util.*;
```

no performance difference

The information for the classes in an imported package is not read in at compile time or runtime unless the class is used in the program. The import statement simply tells the compiler where to locate the classes. There is no performance difference between a specific import and a wildcard import declaration.

Listing 2.3 gives an example of reading multiple inputs from the keyboard. The program reads three numbers and displays their average.

LISTING 2.3 ComputeAverage.java

import class

create a Scanner

```
1  import java.util.Scanner; // Scanner is in the java.util package
2
3  public class ComputeAverage {
4    public static void main(String[] args) {
5      // Create a Scanner object
6      Scanner input = new Scanner(System.in);
7
8      // Prompt the user to enter three numbers
9      System.out.print("Enter three numbers: ");
10       double number1 = input.nextDouble();
11       double number2 = input.nextDouble();
12       double number3 = input.nextDouble();
13
14       // Compute average
15       double average = (number1 + number2 + number3) / 3;
16
17       // Display results
18       System.out.println("The average of " + number1 + " " + number2
19         + " " + number3 + " is " + average);
20     }
21   }
```

read a double

```
Enter three numbers: 1 2 3 ↵Enter
The average of 1.0 2.0 3.0 is 2.0
```

enter input in one line

```
Enter three numbers: 10.5 ↵Enter
11 ↵Enter
11.5 ↵Enter
The average of 10.5 11.0 11.5 is 11.0
```

enter input in multiple lines

The codes for importing the **`Scanner`** class (line 1) and creating a **`Scanner`** object (line 6) are the same as in the preceding example, as well as in all new programs you will write for reading input from the keyboard.

Line 9 prompts the user to enter three numbers. The numbers are read in lines 10–12. You may enter three numbers separated by spaces, then press the *Enter* key, or enter each number followed by a press of the *Enter* key, as shown in the sample runs of this program.

If you entered an input other than a numeric value, a runtime error would occur. In Chapter 12, you will learn how to handle the exception so the program can continue to run.

runtime error

Note

Most of the programs in the early chapters of this book perform three steps— input, process, and output—called *IPO*. Input is receiving input from the user; process is producing results using the input; and output is displaying the results.

IPO

Note

If you use an IDE such as Eclipse or NetBeans, you will get a warning to ask you to close the input for preventing a potential resource leak. Ignore the warning for the time being because the input is automatically closed when your program is terminated. In this case, there will be no resource leaking.

Warning in IDE

Check Point

2.3.1 How do you write a statement to let the user enter a double value from the keyboard? What happens if you entered `5a` when executing the following code?

```
double radius = input.nextDouble();
```

2.3.2 Are there any performance differences between the following two `import` statements?

```
import java.util.Scanner;
import java.util.*;
```

Key Point

2.4 Identifiers

Identifiers are the names that identify the elements such as classes, methods, and variables in a program.

identifiers
identifier naming rules

As you see in Listing 2.3, `ComputeAverage`, `main`, `input`, `number1`, `number2`, `number3`, and so on are the names of things that appear in the program. In programming terminology, such names are called *identifiers*. All identifiers must obey the following rules:

- An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs (`$`).
- An identifier must start with a letter, an underscore (_), or a dollar sign (`$`). It cannot start with a digit.
- An identifier cannot be a reserved word. See Appendix A for a list of reserved words. Reserved words have specific meaning in the Java language. Keywords are reserved words.
- An identifier can be of any length.

For example, `$2`, `ComputeArea`, `area`, `radius`, and `print` are legal identifiers, whereas `2A` and `d+4` are not because they do not follow the rules. The Java compiler detects illegal identifiers and reports syntax errors.

case sensitive

Note

Since Java is case sensitive, `area`, `Area`, and `AREA` are all different identifiers.

descriptive names

Tip

Identifiers are for naming variables, methods, classes, and other items in a program. Descriptive identifiers make programs easy to read. Avoid using abbreviations for identifiers. Using complete words is more descriptive. For example, `numberOfStudents` is better than `numStuds`, `numOfStuds`, or `numOfStudents`. We use descriptive names for complete programs in the text. However, we will occasionally use variable names such as `i`, `j`, `k`, `x`, and `y` in the code snippets for brevity. These names also provide a generic tone to the code snippets.

the $ character

Tip

Do not name identifiers with the `$` character. By convention, the `$` character should be used only in mechanically generated source code.

Check Point

2.4.1 Which of the following identifiers are valid? Which are Java keywords?

```
miles, Test, a++, --a, 4#R, $4, #44, apps
class, public, int, x, y, radius
```

Key Point

2.5 Variables

Variables are used to represent values that may be changed in the program.

why called variables?

As you see from the programs in the preceding sections, variables are used to store values to be used later in a program. They are called variables because their values can be changed. In the program in Listing 2.2, `radius` and `area` are variables of the `double` type. You can assign any numerical value to `radius` and `area`, and the values of `radius` and `area` can be reassigned. For example, in the following code, `radius` is initially `1.0` (line 2) then changed to `2.0` (line 7), and area is set to `3.14159` (line 3) then reset to `12.56636` (line 8).

```
// Compute the first area
radius = 1.0;                                              radius: 1.0
area = radius * radius * 3.14159;                            area: 3.14159
System.out.println("The area is " + area + " for radius " + radius);

// Compute the second area
radius = 2.0;                                              radius: 2.0
area = radius * radius * 3.14159;                            area: 12.56636
System.out.println("The area is " + area + " for radius " + radius);
```

Variables are for representing data of a certain type. To use a variable, you declare it by telling the compiler its name as well as what type of data it can store. The *variable declaration* tells the compiler to allocate appropriate memory space for the variable based on its data type. The syntax for declaring a variable is

```
datatype variableName;
```

Here are some examples of variable declarations:

declare variable

```
int count;              // Declare count to be an integer variable
double radius;          // Declare radius to be a double variable
double interestRate;    // Declare interestRate to be a double variable
```

These examples use the data types `int` and `double`. Later you will be introduced to additional data types, such as `byte`, `short`, `long`, `float`, `char`, and `boolean`.

If variables are of the same type, they can be declared together, as follows:

```
datatype variable1, variable2, ..., variablen;
```

The variables are separated by commas. For example,

```
int i, j, k; // Declare i, j, and k as int variables
```

Variables often have initial values. You can declare a variable and initialize it in one step. Consider, for instance, the following code:

initialize variables

```
int count = 1;
```

This is equivalent to the next two statements:

```
int count;
count = 1;
```

You can also use a shorthand form to declare and initialize variables of the same type together. For example,

```
int i = 1, j = 2;
```

Tip

A variable must be declared before it can be assigned a value. A variable declared in a method must be assigned a value before it can be used.

Whenever possible, declare a variable and assign its initial value in one step. This will make the program easy to read and avoid programming errors.

Every variable has a scope. The *scope of a variable* is the part of the program where the variable can be referenced. The rules that define the scope of a variable will be gradually introduced later in the book. For now, all you need to know is that a variable must be declared and initialized before it can be used.

Check Point

2.5.1 Identify and fix the errors in the following code:

```
public class Test {
  public static void main(String[] args) {
    int i = k + 2;
    System.out.println(i);
  }
}
```

2.6 Assignment Statements and Assignment Expressions

Key Point

An assignment statement assigns a value to a variable. An assignment statement can also be used as an expression in Java.

assignment statement

assignment operator

After a variable is declared, you can assign a value to it by using an *assignment statement*. In Java, the equal sign (`=`) is used as the *assignment operator*. The syntax for assignment statements is as follows:

```
variable = expression;
```

expression

An *expression* represents a computation involving values, variables, and operators that, taking them together, evaluates to a value. In an assignment statement, the expression on the right-hand side of the assignment operator is evaluated, and then the value is assigned to the variable on the left-hand side of the assignment operator. For example, consider the following code:

```
int y = 1;                     // Assign 1 to variable y
double radius = 1.0;           // Assign 1.0 to variable radius
int x = 5 * (3 / 2);           // Assign the value of the expression to x
x = y + 1;                     // Assign the addition of y and 1 to x
double area = radius * radius * 3.14159;     // Compute area
```

You can use a variable in an expression. A variable can also be used in both sides of the `=` operator. For example,

```
x = x + 1;
```

In this assignment statement, the result of `x + 1` is assigned to `x`. If `x` is `1` before the statement is executed, then it becomes `2` after the statement is executed.

To assign a value to a variable, you must place the variable name to the left of the assignment operator. Thus, the following statement is wrong:

```
1 = x; // Wrong
```

Note

In mathematics, `x = 2 * x + 1` denotes an equation. However, in Java, `x = 2 * x + 1` is an assignment statement that evaluates the expression `2 * x + 1` and assigns the result to `x`.

assignment expression

In Java, an assignment statement is essentially an expression that evaluates to the value to be assigned to the variable on the left side of the assignment operator. For this reason, an assignment statement is also known as an *assignment expression*. For example, the following statement is correct:

```
System.out.println(x = 1);
```

which is equivalent to

```
x = 1;
System.out.println(x);
```

If a value is assigned to multiple variables, you can use chained assignments like this:

```
i = j = k = 1;
```

which is equivalent to

```
k = 1;
j = k;
i = j;
```

Note

In an assignment statement, the data type of the variable on the left must be compatible with the data type of the value on the right. For example, **`int x = 1.0`** would be illegal, because the data type of **`x`** is **`int`**. You cannot assign a **`double`** value (**`1.0`**) to an **`int`** variable without using type casting. Type casting will be introduced in Section 2.15.

Check Point

2.6.1 Identify and fix the errors in the following code:

```
public class Test {
  public static void main(String[] args) {
    int i = j = k = 2;
    System.out.println(i + " " + j + " " + k);
  }
}
```

2.7 Named Constants

A named constant is an identifier that represents a permanent value.

Key Point

The value of a variable may change during the execution of a program, but a *named constant,* or simply *constant*, represents permanent data that never changes. A constant is also known as a *final variable* in Java. In our **`ComputeArea`** program, π is a constant. If you use it frequently, you don't want to keep typing **`3.14159`**; instead, you can declare a constant for π. Here is the syntax for declaring a constant:

constant

```
final datatype CONSTANTNAME = value;
```

final keyword

A constant must be declared and initialized in the same statement. The word **`final`** is a Java keyword for declaring a constant. By convention, all letters in a constant are in uppercase. For example, you can declare π as a constant and rewrite Listing 2.2, as in Listing 2.4.

Listing 2.4 ComputeAreaWithConstant.java

```
import java.util.Scanner; // Scanner is in the java.util package

public class ComputeAreaWithConstant {
  public static void main(String[] args) {
    final double PI = 3.14159; // Declare a constant

    // Create a Scanner object
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a radius
    System.out.print("Enter a number for radius: ");
    double radius = input.nextDouble();

    // Compute area
    double area = radius * radius * PI;

    // Display result
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```

benefits of constants

There are three benefits of using constants: (1) you don't have to repeatedly type the same value if it is used multiple times; (2) if you have to change the constant value (e.g., from `3.14` to `3.14159` for `PI`), you need to change it only in a single location in the source code; and (3) a descriptive name for a constant makes the program easy to read.

2.7.1 What are the benefits of using constants? Declare an `int` constant `SIZE` with value `20`.

2.7.2 Translate the following algorithm into Java code:

Step 1: Declare a `double` variable named `miles` with an initial value `100`.

Step 2: Declare a `double` constant named `KILOMETERS_PER_MILE` with value `1.609`.

Step 3: Declare a `double` variable named `kilometers`, multiply `miles` and `KILOMETERS_PER_MILE`, and assign the result to `kilometers`.

Step 4: Display `kilometers` to the console.

What is `kilometers` after Step 4?

2.8 Naming Conventions

Sticking with the Java naming conventions makes your programs easy to read and avoids errors.

Make sure you choose descriptive names with straightforward meanings for the variables, constants, classes, and methods in your program. As mentioned earlier, names are case sensitive. Listed below are the conventions for naming variables, methods, and classes.

name variables and methods

- Use lowercase for variables and methods—for example, the variables `radius` and `area`, and the method `print`. If a name consists of several words, concatenate them into one, making the first word lowercase and capitalizing the first letter of each subsequent word—for example, the variable `numberOfStudents`. This naming style is known as the *camelCase* because the uppercase characters in the name resemble a camel's humps.

name classes

- Capitalize the first letter of each word in a class name—for example, the class names `ComputeArea` and `System`.

name constants

- Capitalize every letter in a constant, and use underscores between words—for example, the constants `PI` and `MAX_VALUE`.

It is important to follow the naming conventions to make your programs easy to read.

Caution

Do not choose class names that are already used in the Java library. For example, since the `System` class is defined in Java, you should not name your class `System`.

2.8.1 What are the naming conventions for class names, method names, constants, and variables? Which of the following items can be a constant, a method, a variable, or a class according to the Java naming conventions?

`MAX_VALUE`, `Test`, `read`, `readDouble`

2.9 Numeric Data Types and Operations

Key Point

*Java has six numeric types for integers and floating-point numbers with operators +, –, *, /, and %.*

Every data type has a range of values. The compiler allocates memory space for each variable or constant according to its data type. Java provides eight primitive data types for numeric values, characters, and Boolean values. This section introduces numeric data types and operators.

Table 2.1 lists the six numeric data types, their ranges, and their storage sizes.

TABLE 2.1 Numeric Data Types

Name	*Range*	*Storage Size*
`byte`	-2^7 to $2^7 - 1$ (−128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15} - 1$ (−32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31} - 1$ (−2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63} - 1$ (i.e., −9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: −3.4028235E + 38 to −1.4E −45 Positive range: 1.4E −45 to 3.4028235E+38 6–9 significant digits	32-bit IEEE 754
`double`	Negative range: −1.7976931348623157E+308 to −4.9E −324 Positive range: 4.9E −324 to 1.7976931348623157E+308 15–17 significant digits	64-bit IEEE 754

byte type

short type

int type

long type

float type

double type

Note

IEEE 754 is a standard approved by the Institute of Electrical and Electronics Engineers for representing floating-point numbers on computers. The standard has been widely adopted. Java uses the 32-bit **IEEE 754** for the `float` type and the 64-bit **IEEE 754** for the `double` type. The **IEEE 754** standard also defines special floating-point values, which are listed in Appendix E.

integer types

Java uses four types for integers: `byte`, `short`, `int`, and `long`. Choose the type that is most appropriate for your variable. For example, if you know an integer stored in a variable is within a range of a byte, declare the variable as a `byte`. For simplicity and consistency, we will use `int` for integers most of the time in this book.

floating-point types

Java uses two types for floating-point numbers: `float` and `double`. The `double` type is twice as big as `float`, so the `double` is known as *double precision*, and `float` as *single precision*. Normally, you should use the `double` type, because it is more accurate than the `float` type.

2.9.1 Reading Numbers from the Keyboard

You know how to use the `nextDouble()` method in the `Scanner` class to read a double value from the keyboard. You can also use the methods listed in Table 2.2 to read a number of the `byte`, `short`, `int`, `long`, and `float` type.

TABLE 2.2 Methods for `Scanner` Objects

Method	*Description*
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.

Here are examples for reading values of various types from the keyboard:

```
 1  Scanner input = new Scanner(System.in);
 2  System.out.print("Enter a byte value: ");
 3  byte byteValue = input.nextByte();
 4
 5  System.out.print("Enter a short value: ");
 6  short shortValue = input.nextShort();
 7
 8  System.out.print("Enter an int value: ");
 9  int intValue = input.nextInt();
10
11  System.out.print("Enter a long value: ");
12  long longValue = input.nextLong();
13
14  System.out.print("Enter a float value: ");
15  float floatValue = input.nextFloat();
```

If you enter a value with an incorrect range or format, a runtime error would occur. For example, if you enter a value `128` for line 3, an error would occur because `128` is out of range for a `byte` type integer.

2.9.2 Numeric Operators

operators +, –, *, /, and %

operands

The operators for numeric data types include the standard arithmetic operators: addition (`+`), subtraction (`–`), multiplication (`*`), division (`/`), and remainder (`%`), as listed in Table 2.3. The *operands* are the values operated by an operator.

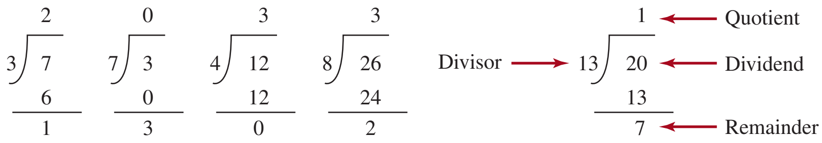
TABLE 2.3 Numeric Operators

Name	*Meaning*	*Example*	*Result*
`+`	Addition	34 + 1	35
`–`	Subtraction	34.0 – 0.1	33.9
`*`	Multiplication	300*30	9000
`/`	Division	1.0 / 2.0	0.5
`%`	Remainder	20 % 3	2

integer division

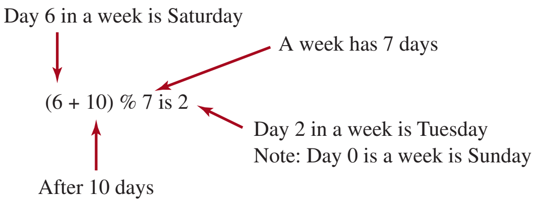
When both operands of a division are integers, the result of the division is the quotient and the fractional part is truncated. For example, `5 / 2` yields `2`, not `2.5`, and `–5 / 2` yields `-2`, not `–2.5`. To perform a floating-point division, one of the operands must be a floating-point number. For example, `5.0 / 2` yields `2.5`.

The `%` operator, known as *remainder*, yields the remainder after division. The operand on the left is the dividend, and the operand on the right is the divisor. Therefore, `7 % 3` yields `1`, `3 % 7` yields `3`, `12 % 4` yields `0`, `26 % 8` yields `2`, and `20 % 13` yields `7`.



The `%` operator is often used for positive integers, but it can also be used with negative integers and floating-point values. The remainder is negative only if the dividend is negative. For example, `-7 % 3` yields `-1`, `-12 % 4` yields `0`, `-26 % -8` yields `-2`, and `20 % -13` yields `7`.

Remainder is very useful in programming. For example, an even number `% 2` is always `0` and a positive odd number `% 2` is always `1`. Thus, you can use this property to determine whether a number is even or odd. If today is Saturday, it will be Saturday again in 7 days. Suppose you and your friends are going to meet in 10 days. What will be the day in 10 days? You can find that the day is Tuesday using the following expression:


Day 6 in a week is Saturday
A week has 7 days
(6 + 10) % 7 is 2
Day 2 in a week is Tuesday
Note: Day 0 is a week is Sunday
After 10 days

The program in Listing 2.5 obtains minutes and remaining seconds from an amount of time in seconds. For example, `500` seconds contains `8` minutes and `20` seconds.

LISTING 2.5 `DisplayTime.java`

```
 1  import java.util.Scanner;
 2
 3  public class DisplayTime {
 4    public static void main(String[] args) {
 5      Scanner input = new Scanner(System.in);
 6      // Prompt the user for input
 7      System.out.print("Enter an integer for seconds: ");
 8      int seconds = input.nextInt();
 9
10      int minutes = seconds / 60; // Find minutes in seconds
11      int remainingSeconds = seconds % 60; // Seconds remaining
12      System.out.println(seconds + " seconds is " + minutes +
13        " minutes and " + remainingSeconds + " seconds");
14    }
15  }
```

import Scanner

create a Scanner

read an integer

divide

remainder

```
Enter an integer for seconds: 500 ↵Enter
500 seconds is 8 minutes and 20 seconds
```

line#	seconds	minutes	remainingSeconds
8	500		
10		8	
11			20

The `nextInt()` method (line 8) reads an integer for `seconds`. Line 10 obtains the minutes using `seconds / 60`. Line 11 (`seconds % 60`) obtains the remaining seconds after taking away the minutes.

unary operator

binary operator

The `+` and – operators can be both unary and binary. A *unary operator* has only one operand; a *binary operator* has two. For example, the `-` operator in `-5` is a unary operator to negate number `5`, whereas the `-` operator in `4 - 5` is a binary operator for subtracting `5` from `4`.

2.9.3 Exponent Operations

Math.pow(a, b) method

The `Math.pow(a, b)` method can be used to compute a^b. The `pow` method is defined in the `Math` class in the Java API. You invoke the method using the syntax `Math.pow(a, b)` (e.g., `Math.pow(2, 3)`), which returns the result of a^b (2^3). Here, `a` and `b` are parameters for the `pow` method and the numbers `2` and `3` are actual values used to invoke the method. For example,

```
System.out.println(Math.pow(2, 3)); // Displays 8.0
System.out.println(Math.pow(4, 0.5)); // Displays 2.0
System.out.println(Math.pow(2.5, 2)); // Displays 6.25
System.out.println(Math.pow(2.5, -2)); // Displays 0.16
```

Chapter 6 introduces more details on methods. For now, all you need to know is how to invoke the `pow` method to perform the exponent operation.

Check Point

2.9.1 Find the largest and smallest `byte`, `short`, `int`, `long`, `float`, and `double`. Which of these data types requires the least amount of memory?

2.9.2 Show the result of the following remainders:

```
 56 %  6
 78 % -4
-34 %  5
-34 % -5
  5 %  1
  1 %  5
```

2.9.3 If today is Tuesday, what will be the day in 100 days?

2.9.4 What is the result of `25 / 4`? How would you rewrite the expression if you wished the result to be a floating-point number?

2.9.5 Show the result of the following code:

```
System.out.println(2 * (5 / 2 + 5 / 2));
System.out.println(2 * 5 / 2 + 2 * 5 / 2);
System.out.println(2 * (5 / 2));
System.out.println(2 * 5 / 2);
```

2.9.6 Are the following statements correct? If so, show the output.

```
System.out.println("25 / 4 is " + 25 / 4);
System.out.println("25 / 4.0 is " + 25 / 4.0);
System.out.println("3 * 2 / 4 is " + 3 * 2 / 4);
System.out.println("3.0 * 2 / 4 is " + 3.0 * 2 / 4);
```

2.9.7 Write a statement to display the result of $2^{3.5}$.

2.9.8 Suppose `m` and `r` are integers. Write a Java expression for mr^2 to obtain a floating-point result.

2.10 Numeric Literals

Key Point

A literal is a constant value that appears directly in a program.

literal

For example, `34` and `0.305` are literals in the following statements:

```
int numberOfYears = 34;
double weight = 0.305;
```

2.10.1 Integer Literals

An integer literal can be assigned to an integer variable as long as it can fit into the variable. A compile error will occur if the literal is too large for the variable to hold. The statement `byte b = 128`, for example, will cause a compile error, because `128` cannot be stored in a variable of the `byte` type. (Note the range for a byte value is from `-128` to `127`.)

An integer literal is assumed to be of the `int` type, whose value is between -2^{31} (-2147483648) and $2^{31}-1$ (2147483647). To denote an integer literal of the `long` type, append the letter `L` or `l` to it. For example, to write integer `2147483648` in a Java program, you have to write it as `2147483648L` or `2147483648l`, because `2147483648` exceeds the range for the `int` value. `L` is preferred because `l` (lowercase `L`) can easily be confused with 1 (the digit one).

Note

binary, octal, and hex literals

By default, an integer literal is a decimal integer number. To denote a binary integer literal, use a leading `0b` or `0B` (zero B); to denote an octal integer literal, use a leading `0` (zero); and to denote a hexadecimal integer literal, use a leading `0x` or `0X` (zero X). For example,

```
System.out.println(0B1111); // Displays 15
System.out.println(07777); // Displays 4095
System.out.println(0XFFFF); // Displays 65535
```

Hexadecimal numbers, binary numbers, and octal numbers will be introduced in Appendix F.

2.10.2 Floating-Point Literals

Floating-point literals are written with a decimal point. By default, a floating-point literal is treated as a `double` type value. For example, `5.0` is considered a `double` value, not a `float` value. You can make a number a `float` by appending the letter `f` or `F`, and you can make a number a `double` by appending the letter `d` or `D`. For example, you can use `100.2f` or `100.2F` for a `float` number, and `100.2d` or `100.2D` for a `double` number.

suffix f or F

suffix d or D

Note

double vs. float

The `double` type values are more accurate than the `float` type values. For example,

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
displays 1.0 / 3.0 is 0.3333333333333333
```

16 digits

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
displays 1.0F / 3.0F is 0.33333334
```

8 digits

A float value has `6–9` numbers of significant digits, and a double value has `15–17` numbers of significant digits.

Note

To improve readability, Java allows you to use underscores to separate two digits in a number literal. For example, the following literals are correct.

```
long value = 232_45_4519;
double amount = 23.24_4545_4519_3415;
```

However, `45_` or `_45` is incorrect. The underscore must be placed between two digits.

underscores in numbers

2.10.3 Scientific Notation

Floating-point literals can be written in scientific notation in the form of $a \times 10^b$. For example, the scientific notation for 123.456 is 1.23456×10^2 and for 0.0123456 is 1.23456×10^{-2}. A special syntax is used to write scientific notation numbers. For example, 1.23456×10^2 is written as `1.23456E2` or `1.23456E+2` and 1.23456×10^{-2} as `1.23456E-2`. `E` (or `e`) represents an exponent, and can be in either lowercase or uppercase.

why called floating-point?

Note

The `float` and `double` types are used to represent numbers with a decimal point. Why are they called *floating-point numbers*? These numbers are stored in scientific notation internally. When a number such as `50.534` is converted into scientific notation, such as `5.0534E+1`, its decimal point is moved (i.e., floated) to a new position.

Check Point

2.10.1 How many accurate digits are stored in a `float` or `double` type variable?

2.10.2 Which of the following are correct literals for floating-point numbers?

`12.3`, `12.3e+2`, `23.4e-2`, `-334.4`, `20.5`, `39F`, `40D`

2.10.3 Which of the following are the same as `52.534`?

`5.2534e+1`, `0.52534e+2`, `525.34e-1`, `5.2534e+0`

2.10.4 Which of the following are correct literals?

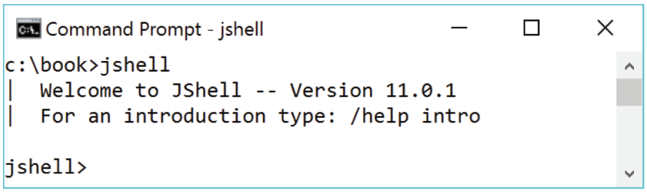
`5_2534e+1, _2534, 5_2, 5_`

2.11 JShell

Key Point

JShell is a command line tool for quickly evaluating an expression and executing a statement.

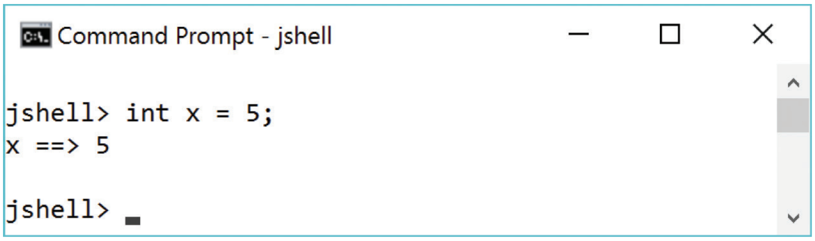
JShell is a command line interactive tool introduced in Java 9. JShell enables you to type a single Java statement and get it executed to see the result right away without having to write a complete class. This feature is commonly known as REPL (Read-Evaluate-Print Loop), which evaluates expressions and executes statements as they are entered and shows the result immediately. To use JShell, you need to install JDK 9 or higher. Make sure that you set the correct path on the Windows environment if you use Windows. Open a Command Window and type jshell to launch JShell as shown in Figure 2.2.



```
c:\book>jshell
|  Welcome to JShell -- Version 11.0.1
|  For an introduction type: /help intro

jshell>
```

FIGURE 2.2 JShell is launched.

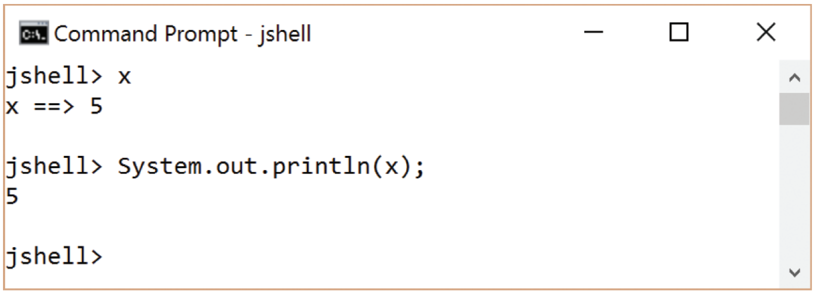
You can enter a Java statement from the jshell prompt. For example, enter `int x = 5`, as shown in Figure 2.3.



```
jshell> int x = 5;
x ==> 5

jshell> _
```

FIGURE 2.3 Enter a Java statement at the jshell command prompt

To print the variable, simply type `x`. Alternatively, you can type `System.out.println(x)`, as shown in Figure 2.4.

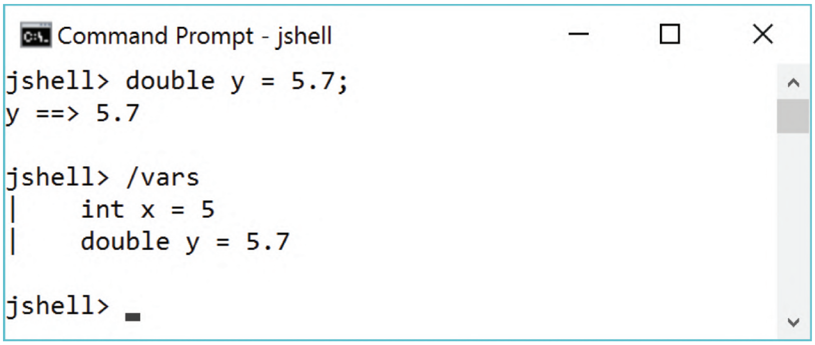


Command Prompt - jshell

```
jshell> x
x ==> 5

jshell> System.out.println(x);
5

jshell>
```

FIGURE 2.4 Print a variable

You can list all the declared variables using the `/vars` command as shown in Figure 2.5.

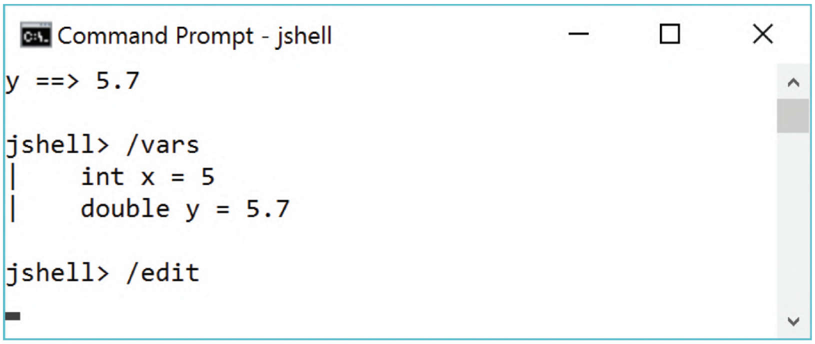


Command Prompt - jshell

```
jshell> double y = 5.7;
y ==> 5.7

jshell> /vars
|    int x = 5
|    double y = 5.7

jshell>
```

FIGURE 2.5 List all variables

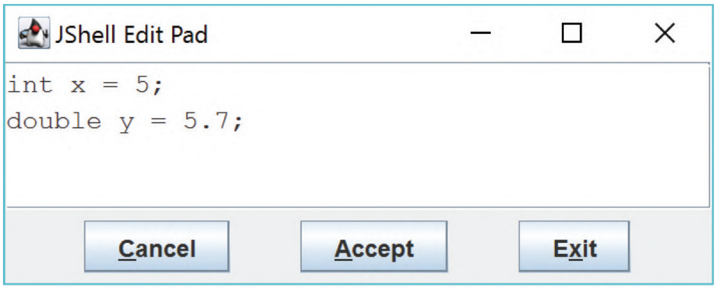
You can use the `/edit` command to edit the code you have entered from the jshell prompt, as shown in Figure 2.6a. This command opens up an edit pane. You can also add/delete the code from the edit pane, as shown in Figure 2.6b. After finishing editing, click the Accept button to make the change in JShell and click the Exit button to exit the edit pane.



Command Prompt - jshell

```
y ==> 5.7

jshell> /vars
|    int x = 5
|    double y = 5.7

jshell> /edit
```

(a)

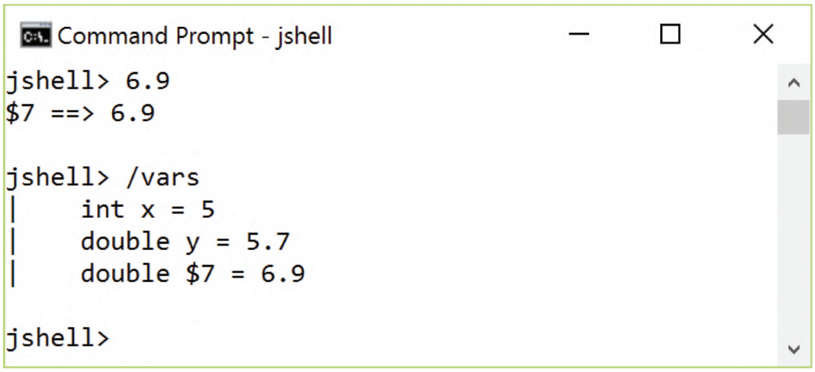


```
int x = 5;
double y = 5.7;
```

(b)

FIGURE 2.6 The /edit command opens up the edit pane

In JShell, if you don't specify a variable for a value, JShell will automatically create a variable for the value. For example, if you type 6.8 from the jshell prompt, you will see variable $7 is automatically created for 6.8, as shown in Figure 2.7.



```
jshell> 6.9
$7 ==> 6.9

jshell> /vars
|    int x = 5
|    double y = 5.7
|    double $7 = 6.9

jshell>
```

FIGURE 2.7 A variable is automatically created for a value.

To exit JShell, enter /exit.
For more information on JShell, see https://docs.oracle.com/en/java/javase/11/jshell/.

Check Point

2.11.1 What does REPL stand for? How do you launch JShell?

2.12 Evaluating Expressions and Operator Precedence

Key Point

Java expressions are evaluated in the same way as arithmetic expressions.

Writing a numeric expression in Java involves a straightforward translation of an arithmetic expression using Java operators. For example, the arithmetic expression

$$\frac{3 + 4x}{5} - \frac{10(y - 5)(a + b + c)}{x} + 9\left(\frac{4}{x} + \frac{9 + x}{y}\right)$$

can be translated into a Java expression as follows:

```
(3 + 4 * x) / 5 - 10 * (y - 5) * (a + b + c) / x +
 9 * (4 / x + (9 + x) / y)
```

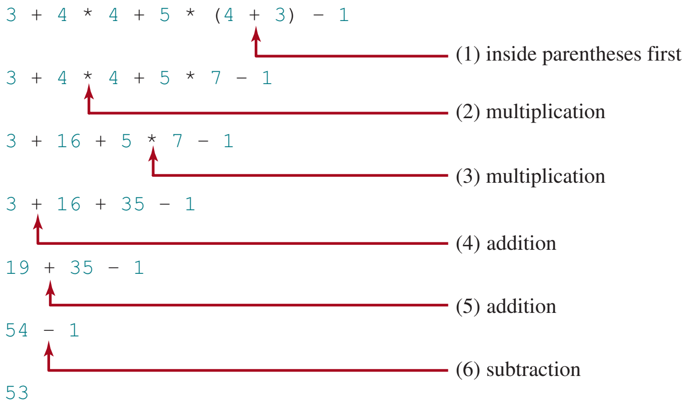
evaluating an expression

Although Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression is the same. Therefore, you can safely apply the arithmetic rule for evaluating a Java expression. Operators contained within pairs of parentheses are evaluated first. Parentheses can be nested, in which case the expression in the inner parentheses is evaluated first. When more than one operator is used in an expression, the following operator precedence rule is used to determine the order of evaluation:

operator precedence rule

- Multiplication, division, and remainder operators are applied first. If an expression contains several multiplication, division, and remainder operators, they are applied from left to right.
- Addition and subtraction operators are applied last. If an expression contains several addition and subtraction operators, they are applied from left to right.

Here is an example of how an expression is evaluated:



Listing 2.6 gives a program that converts a Fahrenheit degree to Celsius using the formula Celsius = $(\frac{5}{9})$(Fahrenheit − 32).

LISTING 2.6 FahrenheitToCelsius.java

```
1   import java.util.Scanner;
2
3   public class FahrenheitToCelsius {
4     public static void main(String[] args) {
5       Scanner input = new Scanner(System.in);
6
7       System.out.print("Enter a degree in Fahrenheit: ");
8       double fahrenheit = input.nextDouble();
9
10      // Convert Fahrenheit to Celsius
11      double celsius = (5.0 / 9) * (fahrenheit - 32);
12      System.out.println("Fahrenheit " + fahrenheit + " is " +
13        celsius + " in Celsius");
14    }
15  }
```

divide

```
Enter a degree in Fahrenheit: 100 ↵Enter
Fahrenheit 100.0 is 37.77777777777778 in Celsius
```

line#	fahrenheit	celsius
8	100	
11		37.77777777777778

integer vs. floating-point division

Be careful when applying division. Division of two integers yields an integer in Java. $\frac{5}{9}$ is coded `5.0 / 9` instead of `5 / 9` in line 11, because `5 / 9` yields `0` in Java.

Check Point

2.12.1 How would you write the following arithmetic expressions in Java?

a. $\frac{4}{3(r + 34)} - 9(a + bc) + \frac{3 + d(2 + a)}{a + bd}$

b. $5.5 \times (r + 2.5)^{2.5+t}$

2.13 Case Study: Displaying the Current Time

Key Point

You can invoke `System.currentTimeMillis()` *to return the current time.*

VideoNote

Use operators / and %

The problem is to develop a program that displays the current time in GMT (Greenwich Mean Time) in the format hour:minute:second, such as 13:19:8.

currentTimeMillis

UNIX epoch

The `currentTimeMillis` method in the `System` class returns the current time in milliseconds elapsed since the time midnight, January 1, 1970 GMT, as shown in Figure 2.8. This time is known as the *UNIX epoch*. The epoch is the point when time starts, and `1970` was the year when the UNIX operating system was formally introduced.

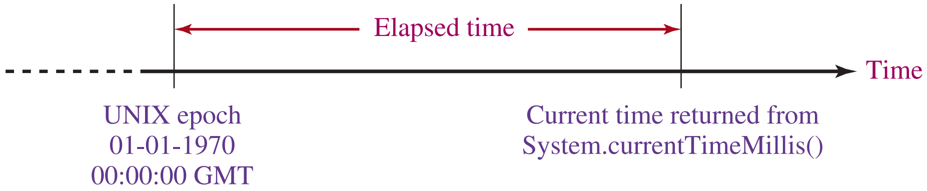


FIGURE 2.8 The `System.currentTimeMillis()` returns the number of milliseconds since the UNIX epoch.

You can use this method to obtain the current time, then compute the current second, minute, and hour as follows:

1. Obtain the total milliseconds since midnight, January 1, 1970, in `totalMilliseconds` by invoking `System.currentTimeMillis()` (e.g., `1203183068328` milliseconds).
2. Obtain the total seconds `totalSeconds` by dividing `totalMilliseconds` by `1000` (e.g., `1203183068328` milliseconds / `1000` = `1203183068` seconds).
3. Compute the current second from `totalSeconds % 60` (e.g., `1203183068` seconds % `60` = `8`, which is the current second).
4. Obtain the total minutes `totalMinutes` by dividing `totalSeconds` by `60` (e.g., `1203183068` seconds / `60` = `20053051` minutes).
5. Compute the current minute from `totalMinutes % 60` (e.g., `20053051` minutes % `60` = `31`, which is the current minute).
6. Obtain the total hours `totalHours` by dividing `totalMinutes` by `60` (e.g., `20053051` minutes / `60` = `334217` hours).
7. Compute the current hour from `totalHours % 24` (e.g., `334217` hours % `24` = `17`, which is the current hour).

Listing 2.7 gives the complete program.

LISTING 2.7 ShowCurrentTime.java

```java
1   public class ShowCurrentTime {
2     public static void main(String[] args) {
3       // Obtain the total milliseconds since midnight, Jan 1, 1970
4       long totalMilliseconds = System.currentTimeMillis();
5
6       // Obtain the total seconds since midnight, Jan 1, 1970
7       long totalSeconds = totalMilliseconds / 1000;
8
9       // Compute the current second in the minute in the hour
10      long currentSecond = totalSeconds % 60;
11
12      // Obtain the total minutes
13      long totalMinutes = totalSeconds / 60;
14
15      // Compute the current minute in the hour
16      long currentMinute = totalMinutes % 60;
17
18      // Obtain the total hours
19      long totalHours = totalMinutes / 60;
20
21      // Compute the current hour
22      long currentHour = totalHours % 24;
23
24      // Display results
25      System.out.println("Current time is " + currentHour + ":"
26        + currentMinute + ":" + currentSecond + " GMT");
27    }
28  }
```

totalMilliseconds

totalSeconds

currentSecond

totalMinutes

currentMinute

totalHours

currentHour

display output

```
Current time is 17:31:8 GMT
```

Line 4 invokes **`System.currentTimeMillis()`** to obtain the current time in milliseconds as a **`long`** value. Thus, all the variables are declared as the long type in this program. The seconds, minutes, and hours are extracted from the current time using the **`/`** and **`%`** operators (lines 6–22).

line# variables	4	7	10	13	16	19	22
totalMilliseconds	1203183068328						
totalSeconds		1203183068					
currentSecond			8				
totalMinutes				20053051			
currentMinute					31		
totalHours						334217	
currentHour							17

In the sample run, a single digit `8` is displayed for the second. The desirable output would be `08`. This can be fixed by using a method that formats a single digit with a prefix `0` (see Programming Exercise 6.37).

The hour displayed in this program is in GMT. Programming Exercise 2.8 enables to display the hour in any time zone.

nanoTime

Java also provides the `System.nanoTime()` method that returns the elapse time in nanoseconds. `nanoTime()` is more precise and accurate than `currentTimeMillis()`.

Check Point

2.13.1 How do you obtain the current second, minute, and hour?

2.14 Augmented Assignment Operators

Key Point

The operators **+**, *−*, *****, **/**, *and* **%** *can be combined with the assignment operator to form augmented operators.*

Very often, the current value of a variable is used, modified, then reassigned back to the same variable. For example, the following statement increases the variable `count` by `1`:

```
count = count + 1;
```

Java allows you to combine assignment and addition operators using an augmented (or compound) assignment operator. For example, the preceding statement can be written as

```
count += 1;
```

addition assignment operator

The `+=` is called the *addition assignment operator*. Table 2.4 shows other augmented assignment operators.

Table 2.4 Augmented Assignment Operators

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`−=`	Subtraction assignment	`i −= 8`	`i = i − 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /= 8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

The augmented assignment operator is performed last after all the other operators in the expression are evaluated. For example,

```
x /= 4 + 5.5 * 1.5;
```

is same as

```
x = x / (4 + 5.5 * 1.5);
```

Caution

There are no spaces in the augmented assignment operators. For example, `+ =` should be `+=`.

Note

Like the assignment operator (`=`), the operators (`+=`, `−=`, `*=`, `/=`, and `%=`) can be used to form an assignment statement as well as an expression. For example, in the following code, `x += 2` is a statement in the first line, and an expression in the second line:

```
x += 2; // Statement
System.out.println(x += 2); // Expression
```

2.14.1 Show the output of the following code:

Check Point

```
double a = 6.5;
a += a + 1;
System.out.println(a);
a = 6;
a /= 2;
System.out.println(a);
```

2.15 Increment and Decrement Operators

The increment operator (++) and decrement operator (−−) are for incrementing and decrementing a variable by 1.

Key Point

increment operator (++)
decrement operator (−−)

The `++` and `--` are two shorthand operators for incrementing and decrementing a variable by `1`. These are handy because that's often how much the value needs to be changed in many programming tasks. For example, the following code increments `i` by `1` and decrements `j` by `1`.

```
int i = 3, j = 3;
i++; // i becomes 4
j--; // j becomes 2
```

postincrement
postdecrement

`i++` is pronounced as "`i` plus plus" and `i--` as "`i` minus minus." These operators are known as *postfix increment* (or *postincrement*) and *postfix decrement* (or *postdecrement*), because the operators `++` and `--` are placed after the variable. These operators can also be placed before the variable. For example,

```
int i = 3, j = 3;
++i; // i becomes 4
--j; // j becomes 2
```

preincrement
predecrement

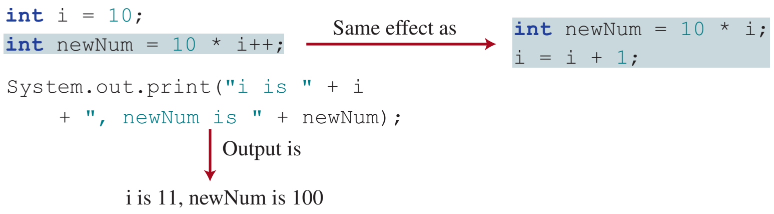
`++i` increments `i` by `1` and `--j` decrements `j` by `1`. These operators are known as *prefix increment* (or *preincrement*) and *prefix decrement* (or predecrement).

As you see, the effect of `i++` and `++i` or `i--` and `--i` are the same in the preceding examples. However, their effects are different when they are used in statements that do more than just increment and decrement. Table 2.5 describes their differences and gives examples.

TABLE 2.5 Increment and Decrement Operators

Operator	*Name*	*Description*	*Example* (assume i = 1)
`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;` // j is 2, i is 2
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;` // j is 1, i is 2
`--var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = --i;` // j is 0, i is 0
`var--`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i--;` // j is 1, i is 0

Here are additional examples to illustrate the differences between the prefix form of `++` (or `--`) and the postfix form of `++` (or `--`). Consider the following code:



```
int i = 10;
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

```
System.out.print("i is " + i
    + ", newNum is " + newNum);
```

Output is

```
i is 11, newNum is 100
```

In this case, `i` is incremented by `1`, then the *old* value of `i` is used in the multiplication. Thus, `newNum` becomes `100`. If `i++` is replaced by `++i`, then it becomes as follows:

```
int i = 10;
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```

```
System.out.print("i is " + i
    + ", newNum is " + newNum);
```

Output is

```
i is 11, newNum is 110
```

`i` is incremented by `1`, and the new value of `i` is used in the multiplication. Thus, `newNum` becomes `110`.

Here is another example:

```
double x = 1.0;
double y = 5.0;
double z = x–– + (++y);
```

After all three lines are executed, `y` becomes `6.0`, `z` becomes `7.0`, and `x` becomes `0.0`.

Operands are evaluated from left to right in Java. The left-hand operand of a binary operator is evaluated before any part of the right-hand operand is evaluated. This rule takes precedence over any other rules that govern expressions. Here is an example:

```
int i = 1;
int k = ++i + i * 3;
```

`++i` is evaluated and returns `2`. When evaluating `i * 3`, `i` is now `2`. Therefore, `k` becomes `8`.

Tip

Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read. Avoid using these operators in expressions that modify multiple variables or the same variable multiple times, such as this one: `int k = ++i + i * 3`.

Check Point

2.15.1 Which of these statements are true?

a. Any expression can be used as a statement.

b. The expression `x++` can be used as a statement.

c. The statement `x = x + 5` is also an expression.

d. The statement `x = y = x = 0` is illegal.

2.15.2 Show the output of the following code:

```
int a = 6;
int b = a++;
System.out.println(a);
System.out.println(b);
a = 6;
b = ++a;
System.out.println(a);
System.out.println(b);
```

2.16 Numeric Type Conversions

Key Point

Floating-point numbers can be converted into integers using explicit casting.

Can you perform binary operations with two operands of different types? Yes. If an integer and a floating-point number are involved in a binary operation, Java automatically converts the integer to a floating-point value. Therefore, `3 * 4.5` is the same as `3.0 * 4.5`.

You can always assign a value to a numeric variable whose type supports a larger range of values; thus, for instance, you can assign a `long` value to a `float` variable. You cannot, however, assign a value to a variable of a type with a smaller range unless you use *type casting*. *Casting* is an operation that converts a value of one data type into a value of another data type. Casting a type with a small range to a type with a larger range is known as *widening a type*. Casting a type with a large range to a type with a smaller range is known as *narrowing a type*. Java will automatically widen a type, but you must narrow a type explicitly.

casting
widening a type
narrowing a type

The syntax for casting a type is to specify the target type in parentheses, followed by the variable's name or the value to be cast. For example, the following statement

```
System.out.println((int)1.7);
```

displays `1`. When a `double` value is cast into an `int` value, the fractional part is truncated.

The following statement

```
System.out.println((double)1 / 2);
```

displays `0.5`, because `1` is cast to `1.0` first, then `1.0` is divided by `2`. However, the statement

```
System.out.println(1 / 2);
```

displays `0`, because `1` and `2` are both integers and the resulting value should also be an integer.

Caution

Casting is necessary if you are assigning a value to a variable of a smaller type range, such as assigning a `double` value to an `int` variable. A compile error will occur if casting is not used in situations of this kind. However, be careful when using casting, as loss of information might lead to inaccurate results.

possible loss of precision

Note

Casting does not change the variable being cast. For example, `d` is not changed after casting in the following code:

```
double d = 4.5;
int i = (int)d; // i becomes 4, but d is still 4.5
```

Note

In Java, an augmented expression of the form `x1 op= x2` is implemented as `x1 = (T)(x1 op x2)`, where `T` is the type for `x1`. Therefore, the following code is correct:

casting in an augmented expression

```
int sum = 0;
sum += 4.5; // sum becomes 4 after this statement
```

`sum += 4.5` is equivalent to `sum = (int)(sum + 4.5)`.

Note

To assign a variable of the `int` type to a variable of the `short` or `byte` type, explicit casting must be used. For example, the following statements have a compile error:

```
int i = 1;
byte b = i; // Error because explicit casting is required
```

However, so long as the integer literal is within the permissible range of the target variable, explicit casting is not needed to assign an integer literal to a variable of the `short` or `byte` type (see Section 2.10, Numeric Literals).

The program in Listing 2.8 displays the sales tax with two digits after the decimal point.

LISTING 2.8 SalesTax.java

casting

```
 1  import java.util.Scanner;
 2
 3  public class SalesTax {
 4    public static void main(String[] args) {
 5      Scanner input = new Scanner(System.in);
 6
 7      System.out.print("Enter purchase amount: ");
 8      double purchaseAmount = input.nextDouble();
 9
10      double tax = purchaseAmount * 0.06;
11      System.out.println("Sales tax is $" + (int)(tax * 100) / 100.0);
12    }
13  }
```

```
Enter purchase amount: 197.55 ↵Enter
Sales tax is $11.85
```

line#	purchaseAmount	tax	Output
8	197.55		
10		11.853	
11			11.85

formatting numbers

Using the input in the sample run, the variable `purchaseAmount` is `197.55` (line 8). The sales tax is `6%` of the purchase, so the `tax` is evaluated as `11.853` (line 10). Note

`tax * 100` is `1185.3`

`(int)(tax * 100)` is `1185`

`(int)(tax * 100) / 100.0` is `11.85`

Thus, the statement in line 11 displays the tax `11.85` with two digits after the decimal point. Note the expression `(int)(tax * 100) / 100.0` rounds down `tax` to two decimal places. If `tax` is `3.456`, `(int)(tax * 100) / 100.0` would be `3.45`. Can it be rounded up to two decimal places? Note any double value `x` can be rounded up to an integer using `(int)(x + 0.5)`. Thus, `tax` can be rounded up to two decimal places using `(int)(tax * 100 + 0.5) / 100.0`.

Check Point

2.16.1 Can different types of numeric values be used together in a computation?

2.16.2 What does an explicit casting from a `double` to an `int` do with the fractional part of the `double` value? Does casting change the variable being cast?

2.16.3 Show the following output:

```
float f = 12.5F;
int i = (int)f;
System.out.println("f is " + f);
System.out.println("i is " + i);
```

2.16.4 If you change `(int)(tax * 100) / 100.0` to `(int)(tax * 100) / 100` in line 11 in Listing 2.8, what will be the output for the input purchase amount of `197.556`?

2.16.5 Show the output of the following code:

```
double amount = 5;
System.out.println(amount / 2);
System.out.println(5 / 2);
```

2.16.6 Write an expression that rounds up a `double` value in variable `d` to an integer.

2.17 Software Development Process

The software development life cycle is a multistage process that includes requirements specification, analysis, design, implementation, testing, deployment, and maintenance.

Key Point

Developing a software product is an engineering process. Software products, no matter how large or how small, have the same life cycle: requirements specification, analysis, design, implementation, testing, deployment, and maintenance, as shown in Figure 2.9.

VideoNote

Software development process

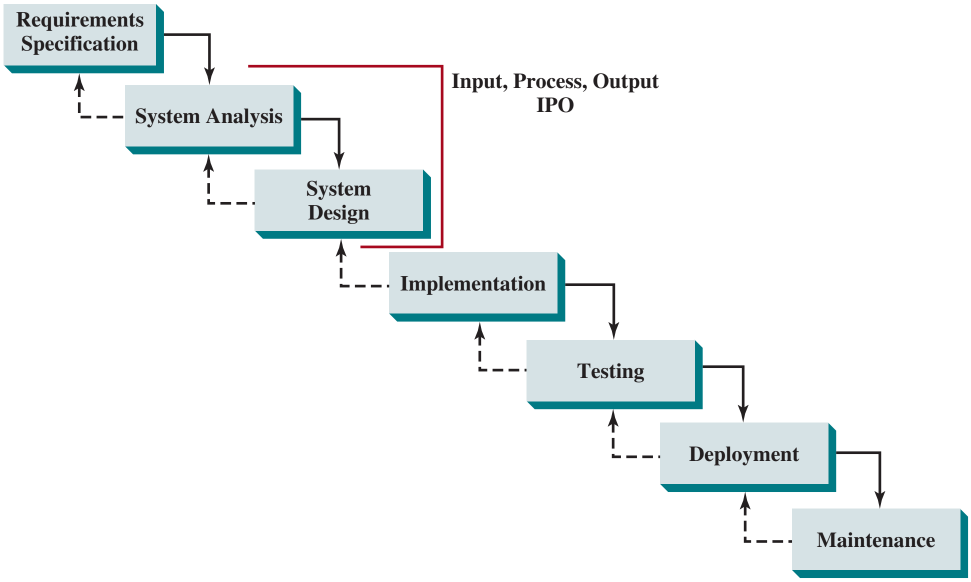


Figure 2.9 At any stage of the software development life cycle, it may be necessary to go back to a previous stage to correct errors or deal with other issues that might prevent the software from functioning as expected.

Requirements specification is a formal process that seeks to understand the problem the software will address, and to document in detail what the software system needs to do. This phase involves close interaction between users and developers. Most of the examples in this book are simple, and their requirements are clearly stated. In the real world, however, problems are not always well defined. Developers need to work closely with their customers (the individuals or organizations that will use the software) and study the problem carefully to identify what the software needs to do.

requirements specification

System analysis seeks to analyze the data flow and to identify the system's input and output. When you perform analysis, it helps to identify what the output is first, then figure out what input data you need in order to produce the output.

system analysis

System design is to design a process for obtaining the output from the input. This phase involves the use of many levels of abstraction to break down the problem into manageable components and design strategies for implementing each component. You can view each component as a subsystem that performs a specific function of the system. The essence of system analysis and design is input, process, and output (IPO).

system design

IOP

Implementation involves translating the system design into programs. Separate programs are written for each component then integrated to work together. This phase requires the use of a programming language such as Java. The implementation involves coding, self-testing, and debugging (that is, finding errors, called *bugs,* in the code).

implementation

Testing ensures the code meets the requirements specification and weeds out bugs. An independent team of software engineers not involved in the design and implementation of the product usually conducts such testing.

testing

deployment

Deployment makes the software available for use. Depending on the type of software, it may be installed on each user's machine, or installed on a server accessible on the Internet.

maintenance

Maintenance is concerned with updating and improving the product. A software product must continue to perform and improve in an ever-evolving environment. This requires periodic upgrades of the product to fix newly discovered bugs and incorporate changes.

VideoNote

Compute loan payments

To see the software development process in action, we will now create a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. For an introductory programming course, we focus on requirements specification, analysis, design, implementation, and testing.

Stage 1: Requirements Specification

The program must satisfy the following requirements:

- It must let the user enter the interest rate, the loan amount, and the number of years for which payments will be made.
- It must compute and display the monthly payment and total payment amounts.

Stage 2: System Analysis

The output is the monthly payment and total payment, which can be obtained using the following formulas:

$$monthlyPayment = \frac{loanAmount \times monthlyInterestRate}{1 - \dfrac{1}{(1 + monthlyInterestRate)^{numberOfYears \times 12}}}$$

$$totalPayment = monthlyPayment \times numberOfYears \times 12$$

Therefore, the input needed for the program is the monthly interest rate, the length of the loan in years, and the loan amount.

Note

The requirements specification says the user must enter the annual interest rate, the loan amount, and the number of years for which payments will be made. During analysis, however, it is possible you may discover that input is not sufficient or some values are unnecessary for the output. If this happens, you can go back and modify the requirements specification.

Note

In the real world, you will work with customers from all walks of life. You may develop software for chemists, physicists, engineers, economists, and psychologists, and of course you will not have (or need) complete knowledge of all these fields. Therefore, you don't have to know how formulas are derived, but given the monthly interest rate, the number of years, and the loan amount, you can compute the monthly payment in this program. You will, however, need to communicate with customers and understand how a mathematical model works for the system.

Stage 3: System Design

During system design, you identify the steps in the program.

Step 3.1. Prompt the user to enter the annual interest rate, the number of years, and the loan amount.

(The interest rate is commonly expressed as a percentage of the principal for a period of one year. This is known as the *annual interest rate*.)

Step 3.2. The input for the annual interest rate is a number in percent format, such as 4.5%. The program needs to convert it into a decimal by dividing it by `100`. To obtain the monthly interest rate from the annual interest rate, divide it by `12`, since a year has 12 months. Thus, to obtain the monthly interest rate in decimal format, you need to divide the annual interest rate in percentage by `1200`. For example, if the annual interest rate is 4.5%, then the monthly interest rate is $4.5/1200 = 0.00375$.

Step 3.3. Compute the monthly payment using the preceding formula.

Step 3.4. Compute the total payment, which is the monthly payment multiplied by `12` and multiplied by the number of years.

Step 3.5. Display the monthly payment and total payment.

Stage 4: Implementation

Implementation is also known as *coding* (writing the code). In the formula, you have to compute $(1 + \texttt{monthlyInterestRate})^{\texttt{numberOfYears} \times 12}$, which can be obtained using `Math.pow(1 + monthlyInterestRate, numberOfYears * 12)`.

Math.pow(a, b) method

Listing 2.9 gives the complete program.

LISTING 2.9 ComputeLoan.java

```
import java.util.Scanner;

public class ComputeLoan {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Enter annual interest rate in percentage, e.g., 7.25
    System.out.print("Enter annual interest rate, e.g., 7.25: ");
    double annualInterestRate = input.nextDouble();

    // Obtain monthly interest rate
    double monthlyInterestRate = annualInterestRate / 1200;

    // Enter number of years
    System.out.print(
      "Enter number of years as an integer, e.g., 5: ");
    int numberOfYears = input.nextInt();

    // Enter loan amount
    System.out.print("Enter loan amount, e.g., 120000.95: ");
    double loanAmount = input.nextDouble();

    // Calculate payment
    double monthlyPayment = loanAmount * monthlyInterestRate / (1
      - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
    double totalPayment = monthlyPayment * numberOfYears * 12;

    // Display results
    System.out.println("The monthly payment is $" +
      (int)(monthlyPayment * 100) / 100.0);
    System.out.println("The total payment is $" +
      (int)(totalPayment * 100) / 100.0);
  }
}
```

import class

create a Scanner

enter interest rate

enter years

enter loan amount

monthlyPayment

totalPayment

casting

casting

```
Enter annual interest rate, for example, 7.25: 5.75 ↵Enter
Enter number of years as an integer, for example, 5: 15 ↵Enter
Enter loan amount, for example, 120000.95: 250000 ↵Enter
The monthly payment is $2076.02
The total payment is $373684.53
```

variables \ line#	10	13	18	22	25	27
`annualInterestRate`	5.75					
`monthlyInterestRate`		0.0047916666666				
`numberOfYears`			15			
`loanAmount`				250000		
`monthlyPayment`					2076.0252175	
`totalPayment`						373684.539

Line 10 reads the annual interest rate, which is converted into the monthly interest rate in line 13.

Choose the most appropriate data type for the variable. For example, `numberOfYears` is best declared as an `int` (line 18), although it could be declared as a `long`, `float`, or `double`. Note `byte` might be the most appropriate for `numberOfYears`. For simplicity, however, the examples in this booktext will use `int` for integer and `double` for floating-point values.

The formula for computing the monthly payment is translated into Java code in lines 25–27.

Casting is used in lines 31 and 33 to obtain a new `monthlyPayment` and `totalPayment` with two digits after the decimal points.

`java.lang` package

The program uses the `Scanner` class, imported in line 1. The program also uses the `Math` class, and you might be wondering why that class isn't imported into the program. The `Math` class is in the `java.lang` package, and all classes in the `java.lang` package are implicitly imported. Therefore, you don't need to explicitly import the `Math` class.

Stage 5: Testing

After the program is implemented, test it with some sample input data and verify whether the output is correct. Some of the problems may involve many cases, as you will see in later chapters. For these types of problems, you need to design test data that cover all cases.

incremental coding and testing

Tip

The system design phase in this example identified several steps. It is a good approach to code and test these steps incrementally by adding them one at a time. This approach, called incremental coding and testing, makes it much easier to pinpoint problems and debug the program.

Check Point

2.17.1 How would you write the following arithmetic expression?

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

2.18 Case Study: Counting Monetary Units

Key Point

This section presents a program that breaks a large amount of money into smaller units.

Suppose you want to develop a program that changes a given amount of money into smaller monetary units. The program lets the user enter an amount as a `double` value representing a total in dollars and cents, and outputs a report listing the monetary equivalent in the maximum number of dollars, quarters, dimes, nickels, and pennies, in this order, to result in the minimum number of coins.

Here are the steps in developing the program:

1. Prompt the user to enter the amount as a decimal number, such as `11.56`.
2. Convert the amount (e.g., `11.56`) into cents (`1156`).
3. Divide the cents by `100` to find the number of dollars. Obtain the remaining cents using the cents remainder `100`.
4. Divide the remaining cents by `25` to find the number of quarters. Obtain the remaining cents using the remaining cents remainder `25`.
5. Divide the remaining cents by `10` to find the number of dimes. Obtain the remaining cents using the remaining cents remainder `10`.
6. Divide the remaining cents by `5` to find the number of nickels. Obtain the remaining cents using the remaining cents remainder `5`.
7. The remaining cents are the pennies.
8. Display the result.

The complete program is given in Listing 2.10.

LISTING 2.10 `ComputeChange.java`

```java
import java.util.Scanner;

public class ComputeChange {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Receive the amount
    System.out.print(
      "Enter an amount in double, for example 11.56: ");
    double amount = input.nextDouble();

    int remainingAmount = (int)(amount * 100);

    // Find the number of one dollars
    int numberOfOneDollars = remainingAmount / 100;
    remainingAmount = remainingAmount % 100;

    // Find the number of quarters in the remaining amount
    int numberOfQuarters = remainingAmount / 25;
    remainingAmount = remainingAmount % 25;

    // Find the number of dimes in the remaining amount
    int numberOfDimes = remainingAmount / 10;
    remainingAmount = remainingAmount % 10;

    // Find the number of nickels in the remaining amount
    int numberOfNickels = remainingAmount / 5;
    remainingAmount = remainingAmount % 5;

    // Find the number of pennies in the remaining amount
    int numberOfPennies = remainingAmount;

```

import class

enter input

dollars

quarters

dimes

nickels

pennies

output

```
34      // Display results
35      System.out.println("Your amount " + amount + " consists of");
36      System.out.println(" " + numberOfOneDollars + " dollars");
37      System.out.println(" " + numberOfQuarters + " quarters ");
38      System.out.println(" " + numberOfDimes + " dimes");
39      System.out.println(" " + numberOfNickels + " nickels");
40      System.out.println(" " + numberOfPennies + " pennies");
41    }
42  }
```

```
Enter an amount in double, for example, 11.56: 11.56 ↵Enter
Your amount 11.56 consists of
      11 dollars
      2 quarters
      0 dimes
      1 nickels
      1 pennies
```

line# / variables	11	13	16	17	20	21	24	25	28	29	32
amount	11.56										
remainingAmount		1156		56		6		6		1	
numberOfOneDollars			11								
numberOfQuarters					2						
numberOfDimes							0				
numberOfNickels									1		
numberOfPennies											1

The variable **`amount`** stores the amount entered from the console (line 11). This variable is not changed, because the amount has to be used at the end of the program to display the results. The program introduces the variable **`remainingAmount`** (line 13) to store the changing remaining amount.

The variable **`amount`** is a **`double`** decimal representing dollars and cents. It is converted to an **`int`** variable **`remainingAmount`**, which represents all the cents. For instance, if **`amount`** is **`11.56`**, then the initial **`remainingAmount`** is **`1156`**. The division operator yields the integer part of the division, so **`1156 / 100`** is **`11`**. The remainder operator obtains the remainder of the division, so **`1156 % 100`** is **`56`**.

The program extracts the maximum number of singles from the remaining amount and obtains a new remaining amount in the variable **`remainingAmount`** (lines 16–17). It then extracts the maximum number of quarters from **`remainingAmount`** and obtains a new **`remainingAmount`** (lines 20–21). Continuing the same process, the program finds the maximum number of dimes, nickels, and pennies in the remaining amount.

loss of precision

One serious problem with this example is the possible loss of precision when casting a **`double`** amount to an **`int remainingAmount`**. This could lead to an inaccurate result. If you try to enter the amount **`10.03`**, **`10.03 * 100`** becomes **`1002.9999999999999`**. You will find that the program displays **`10`** dollars and **`2`** pennies. To fix the problem, enter the amount as an integer value representing cents (see Programming Exercise 2.22).

Check Point

2.18.1 Show the output of Listing 2.10 with the input value **`1.99`**. Why does the program produce an incorrect result for the input 10.03?

2.19 Common Errors and Pitfalls

Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, unintended integer division, and round-off errors.

Key Point

Common Error 1: Undeclared/Uninitialized Variables and Unused Variables

A variable must be declared with a type and assigned a value before using it. A common error is not declaring a variable or initializing a variable. Consider the following code:

```
double interestRate = 0.05;
double interest = interestrate * 45;
```

This code is wrong, because `interestRate` is assigned a value `0.05`; but `interestrate` has not been declared and initialized. Java is case sensitive, so it considers `interestRate` and `interestrate` to be two different variables.

If a variable is declared, but not used in the program, it might be a potential programming error. Therefore, you should remove the unused variable from your program. For example, in the following code, `taxRate` is never used. It should be removed from the code.

```
double interestRate = 0.05;
double taxRate = 0.05;
double interest = interestRate * 45;
System.out.println("Interest is " + interest);
```

If you use an IDE such as Eclipse and NetBeans, you will receive a warning on unused variables.

Common Error 2: Integer Overflow

what is overflow?

Numbers are stored with a limited numbers of digits. When a variable is assigned a value that is too large (*in size*) to be stored, it causes *overflow*. For example, executing the following statement causes overflow, because the largest value that can be stored in a variable of the `int` type is `2147483647`. `2147483648` will be too large for an `int` value:

```
int value = 2147483647 + 1;
// value will actually be -2147483648
```

Likewise, executing the following statement also causes overflow, because the smallest value that can be stored in a variable of the `int` type is `-2147483648`. `-2147483649` is too large in size to be stored in an `int` variable.

```
int value = -2147483648 - 1;
// value will actually be 2147483647
```

Java does not report warnings or errors on overflow, so be careful when working with integers close to the maximum or minimum range of a given type.

what is underflow?

When a floating-point number is too small (i.e., too close to zero) to be stored, it causes *underflow*. Java approximates it to zero, so normally you don't need to be concerned about underflow.

Common Error 3: Round-off Errors

floating-point approximation

A *round-off error*, also called a *rounding error*, is the difference between the calculated approximation of a number and its exact mathematical value. For example, 1/3 is approximately 0.333 if you keep three decimal places, and is 0.3333333 if you keep seven decimal places. Since the number of digits that can be stored in a variable is limited, round-off errors are inevitable. Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy. For example,

```
System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);
```

displays `0.5000000000000001`, not `0.5`, and

```
System.out.println(1.0 - 0.9);
```

displays `0.09999999999999998`, not `0.1`. Integers are stored precisely. Therefore, calculations with integers yield a precise integer result.

Common Error 4: Unintended Integer Division

Java uses the same divide operator, namely `/`, to perform both integer and floating-point division. When two operands are integers, the `/` operator performs an integer division. The result of the operation is an integer. The fractional part is truncated. To force two integers to perform a floating-point division, make one of the integers into a floating-point number. For example, the code in (a) displays that `average` as `1` and the code in (b) displays that `average` as `1.5`.

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2;
System.out.println(average);
```

(a)

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2.0;
System.out.println(average);
```

(b)

Common Pitfall 1: Redundant Input Objects

New programmers often write the code to create multiple input objects for each input. For example, the following code in (a) reads an integer and a double value:

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int v1 = input.nextInt();

Scanner input1 = new Scanner(System.in);
System.out.print("Enter a double value: ");
double v2 = input1.nextDouble();
```

BAD CODE

The code is not good. It creates two input objects unnecessarily and may lead to some subtle errors. You should rewrite the code in (b):

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int v1 = input.nextInt();
System.out.print("Enter a double value: ");
double v2 = input.nextDouble();
```

GOOD CODE

Check Point

2.19.1 Can you declare a variable as `int` and later redeclare it as `double`?

2.19.2 What is an integer overflow? Can floating-point operations cause overflow?

2.19.3 Will overflow cause a runtime error?

2.19.4 What is a round-off error? Can integer operations cause round-off errors? Can floating-point operations cause round-off errors?

KEY TERMS

algorithm, 34
assignment operator (=), 42
assignment statement, 42
`byte` *type*, 45
casting, 59
constant, 43
data type, 35
declare variables, 35

decrement operator (−−), 57
`double` *type*, 45
expression, 42
`final` keyword, 43
`float` *type*, 45
floating-point number, 35
identifier, 40
increment operator (++), 57
incremental coding and testing, 64
`int` *type*, 45
IPO, 39
literal, 48
`long` *type*, 45
narrowing a type, 59
operand, 46
operator, 46
overflow, 67
postdecrement, 57
postincrement, 57
predecrement, 57
preincrement, 57
primitive data type, 35
pseudocode, 34
requirements specification, 61
scope of a variable, 41
`short` *type*, 45
specific import, 38
system analysis, 61
system design, 61
underflow, 67
UNIX epoch, 54
variable, 35
widening a type, 59
wildcard import, 38

CHAPTER SUMMARY

1. *Identifiers* are names for naming elements such as variables, constants, methods, classes, and packages in a program.
2. An identifier is a sequence of characters that consists of letters, digits, underscores (`_`), and dollar signs (`$`). An identifier must start with a letter or an underscore. It cannot start with a digit. An identifier cannot be a reserved word. An identifier can be of any length.
3. *Variables* are used to store data in a program. To declare a variable is to tell the compiler what type of data a variable can hold.
4. There are two types of `import` statements: *specific import* and *wildcard import*. The specific import specifies a single class in the import statement. The wildcard import imports all the classes in a package.
5. In Java, the equal sign (`=`) is used as the *assignment operator.*
6. A variable declared in a method must be assigned a value before it can be used.
7. A *named constant* (or simply a *constant*) represents permanent data that never changes.
8. A named constant is declared by using the keyword `final`.
9. Java provides four integer types (`byte`, `short`, `int`, and `long`) that represent integers of four different sizes.
10. Java provides two *floating-point types* (`float` and `double`) that represent floating-point numbers of two different precisions.
11. Java provides *operators* that perform numeric operations: `+` (addition), `–` (subtraction), `*` (multiplication), `/` (division), and `%` (remainder).
12. Integer arithmetic (`/`) yields an integer result.
13. The numeric operators in a Java expression are applied the same way as in an arithmetic expression.

14. Java provides the augmented assignment operators `+=` (addition assignment), `-=` (subtraction assignment), `*=` (multiplication assignment), `/=` (division assignment), and `%=` (remainder assignment).

15. The *increment operator* (`++`) and the *decrement operator* (`--`) increment or decrement a variable by `1`.

16. When evaluating an expression with values of mixed types, Java automatically converts the operands to appropriate types.

17. You can explicitly convert a value from one type to another using the `(type)value` notation.

18. *Casting* a variable of a type with a small range to a type with a larger range is known as *widening a type*.

19. Casting a variable of a type with a large range to a type with a smaller range is known as *narrowing a type.*

20. Widening a type can be performed automatically without explicit casting. Narrowing a type must be performed explicitly.

21. In computer science, midnight of January 1, 1970, is known as the *UNIX epoch.*

Quiz

Answer the quiz for this chapter online at the Companion Website.

MyProgrammingLab™

Programming Exercises

Debugging Tip

learn from examples

The compiler usually gives a reason for a syntax error. If you don’t know how to correct it, compare your program closely, character by character, with similar examples in the text.

Pedagogical Note

document analysis and design

Instructors may ask you to document your analysis and design for selected exercises. Use your own words to analyze the problem, including the input, output, and what needs to be computed, and describe how to solve the problem in pseudocode.

Pedagogical Note

even-numbered programming exercises

The solution to most even-numbered programming exercises are provided to students. These exercises serve as additional examples for a variety of programs. To maximize the benefits of these solutions, students should first attempt to complete the even-numbered exercises and then compare their solutions with the solutions provided in the book. Since the book provides a large number of programming exercises, it is sufficient if you can complete all even-numbered programming exercises.

Sections 2.2–2.13

2.1 (*Convert Celsius to Fahrenheit*) Write a program that reads a Celsius degree in a `double` value from the console, then converts it to Fahrenheit, and displays the result. The formula for the conversion is as follows:

```
fahrenheit = (9 / 5) * celsius + 32
```

Hint: In Java, `9 / 5` is `1`, but `9.0 / 5` is `1.8`.

Here is a sample run:

```
Enter a degree in Celsius: 43.5 ↵Enter
43.5 Celsius is 110.3 Fahrenheit
```

2.2 (*Compute the volume of a cylinder*) Write a program that reads in the radius and length of a cylinder and computes the area and volume using the following formulas:

```
area = radius * radius * π
volume = area * length
```

Here is a sample run:

```
Enter the radius and length of a cylinder: 5.5 12 ↵Enter
The area is 95.0331
The volume is 1140.4
```

2.3 (*Convert feet into meters*) Write a program that reads a number in feet, converts it to meters, and displays the result. One foot is `0.305` meter. Here is a sample run:

```
Enter a value for feet: 16.5 ↵Enter
16.5 feet is 5.0325 meters
```

2.4 (*Convert pounds into kilograms*) Write a program that converts pounds into kilograms. The program prompts the user to enter a number in pounds, converts it to kilograms, and displays the result. One pound is `0.454` kilogram. Here is a sample run:

```
Enter a number in pounds: 55.5 ↵Enter
55.5 pounds is 25.197 kilograms
```

***2.5** (*Financial application: calculate tips*) Write a program that reads the subtotal and the gratuity rate, then computes the gratuity and total. For example, if the user enters `10` for subtotal and `15%` for gratuity rate, the program displays `$1.5` as gratuity and `$11.5` as total. Here is a sample run:

```
Enter the subtotal and a gratuity rate: 10 15 ↵Enter
The gratuity is $1.5 and total is $11.5
```

****2.6** (*Sum the digits in an integer*) Write a program that reads an integer between `0` and `1000` and adds all the digits in the integer. For example, if an integer is `932`, the sum of all its digits is `14`.

Hint: Use the `%` operator to extract digits, and use the `/` operator to remove the extracted digit. For instance, `932 % 10 = 2` and `932 / 10 = 93`.

Here is a sample run:

```
Enter a number between 0 and 1000: 999 ↵Enter
The sum of the digits is 27
```

***2.7** (*Find the number of years*) Write a program that prompts the user to enter the minutes (e.g., 1 billion), and displays the maximum number of years and remaining days for the minutes. For simplicity, assume that a year has **`365`** days. Here is a sample run:

```
Enter the number of minutes: 1000000000 ↵Enter
1000000000 minutes is approximately 1902 years and 214 days
```

***2.8** (*Current time*) Listing 2.7, ShowCurrentTime.java, gives a program that displays the current time in GMT. Revise the program so it prompts the user to enter the time zone offset to GMT and displays the time in the specified time zone. Here is a sample run:

```
Enter the time zone offset to GMT: -5 ↵Enter
The current time is 4:50:34
```

2.9 (*Physics: acceleration*) Average acceleration is defined as the change of velocity divided by the time taken to make the change, as given by the following formula:

$$a = \frac{v_1 - v_0}{t}$$

Write a program that prompts the user to enter the starting velocity v_0 in meters/second, the ending velocity v_1 in meters/second, and the time span t in seconds, then displays the average acceleration. Here is a sample run:

```
Enter v0, v1, and t: 5.5 50.9 4.5
The average acceleration is 10.0889
```

2.10 (*Science: calculating energy*) Write a program that calculates the energy needed to heat water from an initial temperature to a final temperature. Your program should prompt the user to enter the amount of water in kilograms and the initial and final temperatures of the water. The formula to compute the energy is

```
Q = M * (finalTemperature – initialTemperature) * 4184
```

where **`M`** is the weight of water in kilograms, initial and final temperatures are in degrees Celsius, and energy **`Q`** is measured in joules. Here is a sample run:

```
Enter the amount of water in kilograms: 55.5 ↵Enter

Enter the initial temperature: 3.5 ↵Enter

Enter the final temperature: 10.5 ↵Enter

The energy needed is 1625484.0
```

2.11 (*Population projection*) Rewrite Programming Exercise 1.11 to prompt the user to enter the number of years and display the population after the number of years. Use the hint in Programming Exercise 1.11 for this program. Here is a sample run of the program:

```
Enter the number of years: 5 ↵Enter
The population in 5 years is 325932969
```

2.12 (*Physics: finding runway length*) Given an airplane's acceleration a and take-off speed v, you can compute the minimum runway length needed for an airplane to take off using the following formula:

$$\text{length} = \frac{v^2}{2a}$$

Write a program that prompts the user to enter v in meters/second (m/s) and the acceleration a in meters/second squared (m/s^2), then, displays the minimum runway length.

```
Enter speed and acceleration: 60 3.5 ↵Enter
The minimum runway length for this airplane is 514.286
```

****2.13** (*Financial application: compound value*) Suppose you save **`$100`** *each* month into a savings account with an annual interest rate of 5%. Thus, the monthly interest rate is $0.05/12 = 0.00417$. After the first month, the value in the account becomes

```
100 * (1 + 0.00417) = 100.417
```

After the second month, the value in the account becomes

```
(100 + 100.417) * (1 + 0.00417) = 201.252
```

After the third month, the value in the account becomes

```
(100 + 201.252) * (1 + 0.00417) = 302.507
```

and so on.

Write a program that prompts the user to enter a monthly saving amount and displays the account value after the sixth month. (In Programming Exercise 5.30, you will use a loop to simplify the code and display the account value for any month.)

```
Enter the monthly saving amount: 100 ↵Enter
After the sixth month, the account value is $608.81
```

***2.14** (*Health application: computing BMI*) Body Mass Index (BMI) is a measure of health on weight. It can be calculated by taking your weight in kilograms and dividing, by the square of your height in meters. Write a program that prompts the user to enter a weight in pounds and height in inches and displays the BMI. Note one pound is **`0.45359237`** kilograms and one inch is **`0.0254`** meters. Here is a sample run:

VideoNote
Compute BMI

```
Enter weight in pounds: 95.5 ↵Enter

Enter height in inches: 50 ↵Enter

BMI is 26.8573
```

2.15 (*Geometry: distance of two points*) Write a program that prompts the user to enter two points **`(x1, y1)`** and **`(x2, y2)`** and displays their distance. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Note you can use **`Math.pow(a, 0.5)`** to compute $\sqrt{a}$. Here is a sample run:

```
Enter x1 and y1: 1.5 -3.4 ↵Enter

Enter x2 and y2: 4 5 ↵Enter

The distance between the two points is 8.764131445842194
```

2.16 (*Geometry: area of a hexagon*) Write a program that prompts the user to enter the side of a hexagon and displays its area. The formula for computing the area of a hexagon is

$$\text{Area} = \frac{3\sqrt{3}}{2}s^2,$$

where s is the length of a side. Here is a sample run:

```
Enter the length of the side: 5.5 ↵Enter
The area of the hexagon is 78.5918
```

***2.17** (*Science: wind-chill temperature*) How cold is it outside? The temperature alone is not enough to provide the answer. Other factors including wind speed, relative humidity, and sunshine play important roles in determining coldness outside. In 2001, the National Weather Service (NWS) implemented the new wind-chill temperature to measure the coldness using temperature and wind speed. The formula is

$$t_{wc} = 35.74 + 0.6215t_a - 35.75v^{0.16} + 0.4275t_a v^{0.16}$$

where t_a is the outside temperature measured in degrees Fahrenheit, v is the speed measured in miles per hour, and t_{wc} is the wind-chill temperature. The formula cannot be used for wind speeds below 2 mph or temperatures below −58°F or above 41°F.

Write a program that prompts the user to enter a temperature between −58°F and 41°F and a wind speed greater than or equal to `2` then displays the wind-chill temperature. Use `Math.pow(a, b)` to compute $v^{0.16}$. Here is a sample run:

```
Enter the temperature in Fahrenheit between −58°F and 41°F:
5.3 ↵Enter
Enter the wind speed (>= 2) in miles per hour: 6 ↵Enter
The wind chill index is −5.56707
```

2.18 (*Print a table*) Write a program that displays the following table. Cast floating-point numbers into integers.

```
a    b    pow(a, b)
1    2    1
2    3    8
3    4    81
4    5    1024
5    6    15625
```

***2.19** (*Geometry: area of a triangle*) Write a program that prompts the user to enter three points, `(x1, y1)`, `(x2, y2)`, and `(x3, y3)`, of a triangle then displays its area. The formula for computing the area of a triangle is

$$s = (\text{side1} + \text{side2} + \text{side3})/2;$$

$$\text{area} = \sqrt{s(s - \text{side1})(s - \text{side2})(s - \text{side3})}$$

Here is a sample run:

```
Enter the coordinates of three points separated by spaces
like x1 y1 x2 y2 x3 y3: 1.5 −3.4 4.6 5 9.5 −3.4 ↵Enter
The area of the triangle is 33.6
```

Sections 2.13–2.18

***2.20** (*Financial application: calculate interest*) If you know the balance and the annual percentage interest rate, you can compute the interest on the next monthly payment using the following formula:

$$\text{interest} = \text{balance} \times (\texttt{annualInterestRate}/1200)$$

Write a program that reads the balance and the annual percentage interest rate and displays the interest for the next month. Here is a sample run:

```
Enter balance and interest rate (e.g., 3 for 3%): 1000 3.5 ↵Enter
The interest is 2.91667
```

***2.21** (*Financial application: calculate future investment value*) Write a program that reads in investment amount, annual interest rate, and number of years and displays the future investment value using the following formula:

$$\texttt{futureInvestmentValue} =$$
$$\texttt{investmentAmount} \times (1 + \texttt{monthlyInterestRate})^{\texttt{numberOfYears}*12}$$

For example, if you enter amount `1000`, annual interest rate `3.25%`, and number of years `1`, the future investment value is `1032.98`.

Here is a sample run:

```
Enter investment amount: 1000.56 ↵Enter
Enter annual interest rate in percentage: 4.25 ↵Enter
Enter number of years: 1 ↵Enter
Future value is $1043.92
```

***2.22** (*Financial application: monetary units*) Rewrite Listing 2.10, ComputeChange.java, to fix the possible loss of accuracy when converting a `double` value to an `int` value. Enter the input as an integer whose last two digits represent the cents. For example, the input `1156` represents `11` dollars and `56` cents.

***2.23** (*Cost of driving*) Write a program that prompts the user to enter the distance to drive, the fuel efficiency of the car in miles per gallon, and the price per gallon then displays the cost of the trip. Here is a sample run:

```
Enter the driving distance: 900.5 ↵Enter
Enter miles per gallon: 25.5 ↵Enter
Enter price per gallon: 3.55 ↵Enter
The cost of driving is $125.36
```

Note

More than 200 additional programming exercises with solutions are provided to the instructors on the Instructor Resource Website.

CHAPTER

3

SELECTIONS

Objectives

- To declare **`boolean`** variables and write Boolean expressions using relational operators (§3.2).
- To implement selection control using one-way **`if`** statements (§3.3).
- To implement selection control using two-way **`if-else`** statements (§3.4).
- To implement selection control using nested **`if`** and multi-way **`if`** statements (§3.5).
- To avoid common errors and pitfalls in **`if`** statements (§3.6).
- To generate random numbers using the **`Math.random()`** method (§3.7).
- To program using selection statements for a variety of examples (**`SubtractionQuiz`**, **`BMI`**, **`ComputeTax`**) (§§3.7–3.9).
- To combine conditions using logical operators (**`!`**, **`&&`**, **`||`**, and **`^`**) (§3.10).
- To program using selection statements with combined conditions (**`LeapYear`**, **`Lottery`**) (§§3.11 and 3.12).
- To implement selection control using **`switch`** statements (§3.13).
- To write expressions using the conditional operator (§3.14).
- To examine the rules governing operator precedence and associativity (§3.15).
- To apply common techniques to debug errors (§3.16).

3.1 Introduction

Key Point

The program can decide which statements to execute based on a condition.

problem

If you enter a negative value for `radius` in Listing 2.2, ComputeAreaWithConsoleInput.java, the program displays an invalid result. If the radius is negative, you don't want the program to compute the area. How can you deal with this situation?

selection statements

Like all high-level programming languages, Java provides *selection statements*: statements that let you choose actions with alternative courses. You can use the following selection statement to replace lines 12–17 in Listing 2.2:

```
if (radius < 0) {
  System.out.println("Incorrect input");
}
else {
  double area = radius * radius * 3.14159;
  System.out.println("Area is " + area);
}
```

Boolean expression

Boolean value

Selection statements use conditions that are Boolean expressions. A *Boolean expression* is an expression that evaluates to a *Boolean value*: `true` or `false`. We now introduce the `boolean` type and relational operators.

3.2 `boolean` Data Type, Values, and Expressions

Key Point

The `boolean` data type declares a variable with the value either `true` or `false`.

`boolean` data type

relational operators

How do you compare two values, such as whether a radius is greater than `0`, equal to `0`, or less than `0`? Java provides six *relational operators* (also known as *comparison operators*), shown in Table 3.1, which can be used to compare two values (assume radius is `5` in the table).

TABLE 3.1 Relational Operators

Java Operator	*Mathematics Symbol*	*Name*	*Example (radius is 5)*	*Result*
`<`	<	Less than	`radius < 0`	`false`
`<=`	≤	Less than or equal to	`radius <= 0`	`false`
`>`	>	Greater than	`radius > 0`	`true`
`>=`	≥	Greater than or equal to	`radius >= 0`	`true`
`==`	=	Equal to	`radius == 0`	`false`
`!=`	≠	Not equal to	`radius != 0`	`true`

`== vs. =`

Caution

The equality testing operator is two equal signs (`==`), not a single equal sign (`=`). The latter symbol is for assignment.

The result of the comparison is a Boolean value: `true` or `false`. For example, the following statement displays `true`:

```
double radius = 1;
System.out.println(radius > 0);
```

Boolean variable

A variable that holds a Boolean value is known as a *Boolean variable*. The `boolean` data type is used to declare Boolean variables. A `boolean` variable can hold one of the two values: `true` or `false`. For example, the following statement assigns `true` to the variable `lightsOn`:

```
boolean lightsOn = true;
```

Boolean literals

`true` and `false` are literals, just like a number such as `10`. They are not keywords, but are reserved words and cannot be used as identifiers in the program.

VideoNote

Program addition quiz

Suppose you want to develop a program to let a first-grader practice addition. The program randomly generates two single-digit integers, `number1` and `number2`, and displays to the student a question such as “What is 1 + 7?, ” as shown in the sample run in Listing 3.1. After the student types the answer, the program displays a message to indicate whether it is true or false.

There are several ways to generate random numbers. For now, generate the first integer using `System.currentTimeMillis() % 10` (i.e., the last digit in the current time) and the second using `System.currentTimeMillis() / 10 % 10` (i.e., the second last digit in the current time). Listing 3.1 gives the program. Lines 5–6 generate two numbers, `number1` and `number2`. Line 14 obtains an answer from the user. The answer is graded in line 18 using a Boolean expression `number1 + number2 == answer`.

LISTING 3.1 AdditionQuiz.java

```
 1  import java.util.Scanner;
 2
 3  public class AdditionQuiz {
 4    public static void main(String[] args) {
 5      int number1 = (int)(System.currentTimeMillis() % 10);
 6      int number2 = (int)(System.currentTimeMillis() / 10 % 10);
 7
 8      // Create a Scanner
 9      Scanner input = new Scanner(System.in);
10
11      System.out.print(
12        "What is " + number1 + " + " + number2 + "? ");
13
14      int answer = input.nextInt();
15
16      System.out.println(
17        number1 + " + " + number2 + " = " + answer + " is " +
18        (number1 + number2 == answer));
19    }
20  }
```

generate number1
generate number2
show question
receive answer
display result

```
What is 1 + 7? 8 ↵Enter
1 + 7 = 8 is true
```

```
What is 4 + 8? 9 ↵Enter
4 + 8 = 9 is false
```

line#	number1	number2	answer	output
5	4			
6		8		
14			9	
16				4 + 8 = 9 is false

Check Point

3.2.1 List six relational operators.

3.2.2 Assuming `x` is `1`, show the result of the following Boolean expressions:

```
(x > 0)
(x < 0)
(x != 0)
(x >= 0)
(x != 1)
```

3.2.3 Can the following conversions involving casting be allowed? Write a test program to verify it.

```
boolean b = true;
i = (int)b;

int i = 1;
boolean b = (boolean)i;
```

3.3 `if` Statements

Key Point

An `if` statement is a construct that enables a program to specify alternative paths of execution.

why `if` statement?

The preceding program displays a message such as "6 + 2 = 7 is false." If you wish the message to be "6 + 2 = 7 is incorrect," you have to use a selection statement to make this minor change.

Java has several types of selection statements: one-way `if` statements, two-way `if-else` statements, nested `if` statements, multi-way `if-else` statements, `switch` statements, and conditional operators.

A one-way `if` statement executes an action if and only if the condition is `true`. The syntax for a one-way `if` statement is as follows:

`if` statement?

```
if (boolean-expression) {
  statement(s);
}
```

flowchart

The flowchart in Figure 3.1a illustrates how Java executes the syntax of an `if` statement. A *flowchart* is a diagram that describes an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. Process operations are represented in these boxes, and the arrows connecting them represent the flow of control. A diamond box denotes a Boolean condition, and a rectangle box represents statements.

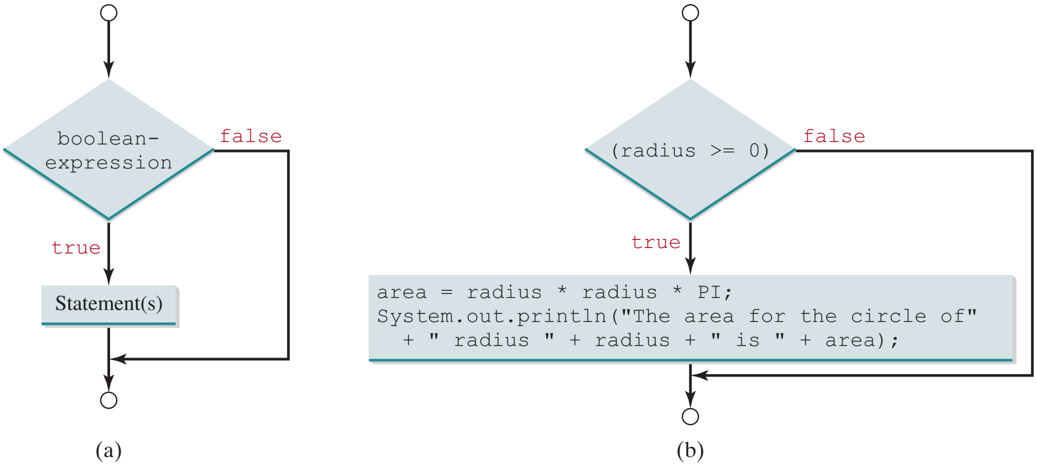

boolean-expression
false
true
Statement(s)
(a)
(radius >= 0)
false
true
area = radius * radius * PI;
System.out.println("The area for the circle of"
  + " radius " + radius + " is " + area);
(b)

FIGURE 3.1 An `if` statement executes statements if the `boolean-expression` evaluates to `true`.

If the `boolean-expression` evaluates to `true`, the statements in the block are executed. As an example, see the following code:

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area for the circle of radius " +
    radius + " is " + area);
}
```

The flowchart of the preceding statement is shown in Figure 3.1b. If the value of `radius` is greater than or equal to `0`, then the `area` is computed and the result is displayed; otherwise, the two statements in the block will not be executed.

The `boolean-expression` is enclosed in parentheses. For example, the code in (a) is wrong. It should be corrected, as shown in (b).

```
if  i > 0  {
  System.out.println("i is positive");
}
```

(a) Wrong

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(b) Correct

The block braces can be omitted if they enclose a single statement. For example, the following statements are equivalent:

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(a)

Equivalent

```
if (i > 0)
  System.out.println("i is positive");
```

(b)

Caution

Omitting braces makes the code shorter, but it is prone to errors. It is a common mistake to forget the braces when you go back to modify the code that omits the braces.

Omitting braces or not

Listing 3.2 gives a program that prompts the user to enter an integer. If the number is a multiple of `5`, the program displays `HiFive`. If the number is divisible by `2`, it displays `HiEven`.

LISTING 3.2 `SimpleIfDemo.java`

```
import java.util.Scanner;

public class SimpleIfDemo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    int number = input.nextInt();

    if (number % 5 == 0)
      System.out.println("HiFive");

    if (number % 2 == 0)
      System.out.println("HiEven");
  }
}
```

enter input

check 5

check even

```
Enter an integer: 4 ↵Enter
HiEven
```

```
Enter an integer: 30 ↵Enter
HiFive
HiEven
```

The program prompts the user to enter an integer (lines 6–7) and displays `HiFive` if it is divisible by `5` (lines 9–10) and `HiEven` if it is divisible by `2` (lines 12–13).

Check Point

3.3.1 Write an `if` statement that assigns `1` to `x` if `y` is greater than `0`.

3.3.2 Write an `if` statement that increases pay by 3% if `score` is greater than `90`.

3.3.3 What is wrong in the following code?

```
if radius >= 0
{
  area = radius * radius * PI;
  System.out.println("The area for the circle of " +
    " radius " + radius + " is " + area);
}
```

Key Point

3.4 Two-Way `if-else` Statements

An `if-else` statement decides the execution path based on whether the condition is true or false.

A one-way `if` statement performs an action if the specified condition is `true`. If the condition is `false`, nothing is done. But what if you want to take alternative actions when the condition is `false`? You can use a two-way `if-else` statement. The actions that a two-way `if-else` statement specifies differ based on whether the condition is `true` or `false`.

Here is the syntax for a two-way `if-else` statement:

```
if (boolean-expression) {
  statement(s)-for-the-true-case;
}
else {
  statement(s)-for-the-false-case;
}
```

The flowchart of the statement is shown in Figure 3.2.

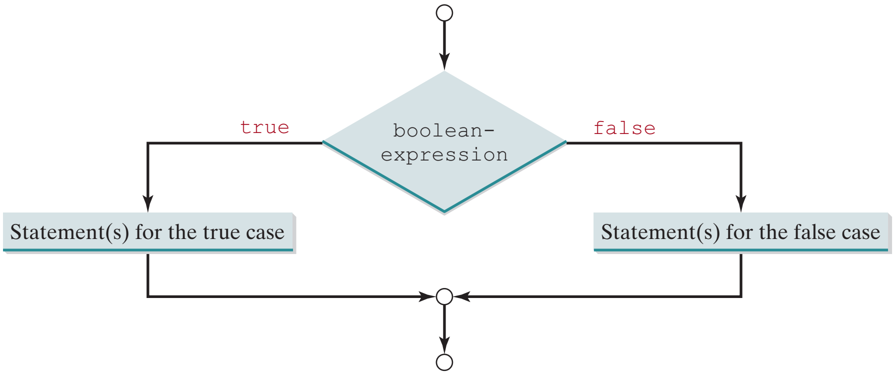

true
boolean-expression
false
Statement(s) for the true case
Statement(s) for the false case

FIGURE 3.2 An `if-else` statement executes statements for the true case if the `boolean-expression` evaluates to `true`; otherwise, statements for the `false` case are executed.

If the `boolean-expression` evaluates to `true`, the statement(s) for the true case are executed; otherwise, the statement(s) for the `false` case are executed. For example, consider the following code:

two-way `if-else` statement

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area for the circle of radius " +
    radius + " is " + area);
}
else {
  System.out.println("Negative input");
}
```

If `radius >= 0` is `true`, `area` is computed and displayed; if it is `false`, the message `"Negative input"` is displayed.

As usual, the braces can be omitted if there is only one statement within them. The braces enclosing the `System.out.println("Negative input")` statement can therefore be omitted in the preceding example.

Here is another example of using the `if-else` statement. The example checks whether a number is even or odd, as follows:

```
if (number % 2 == 0)
  System.out.println(number + " is even.");
else
  System.out.println(number + " is odd.");
```

Check Point

3.4.1 Write an `if` statement that increases `pay` by 3% if `score` is greater than `90`, otherwise increases `pay` by 1%.

3.4.2 What is the output of the code in (a) and (b) if `number` is `30`? What if `number` is `35`?

```
if (number % 2 == 0)
  System.out.println(number
    + "is even.");

System.out.println(number
    + "is odd");
```

(a)

```
if (number % 2 == 0)
  System.out.println(number
    + "is even.");
else
System.out.println(number
    + "is odd");
```

(b)

3.5 Nested `if` and Multi-Way `if-else` Statements

Key Point

An `if` statement can be inside another `if` statement to form a nested `if` statement.

The statement in an `if` or `if-else` statement can be any legal Java statement, including another `if` or `if-else` statement. The inner `if` statement is said to be *nested* inside the outer `if` statement. The inner `if` statement can contain another `if` statement; in fact, there is no limit to the depth of the nesting. For example, the following is a nested `if` statement:

nested `if` statement

```
if (i > k) {
  if (j > k)
    System.out.println("i and j are greater than k");
}
else
  System.out.println("i is less than or equal to k");
```

The `if (j > k)` statement is nested inside the `if (i > k)` statement.

The nested `if` statement can be used to implement multiple alternatives. The statement given in Figure 3.3a, for instance, prints a letter grade according to the score, with multiple alternatives.

```
if (score >= 90)
  System.out.print("A");
else
  if (score >= 80)
    System.out.print("B");
  else
    if (score >= 70)
      System.out.print("C");
    else
      if (score >= 60)
        System.out.print("D");
      else
        System.out.print("F");
```

(a)

Equivalent

This is better

```
if (score >= 90)
  System.out.print("A");
else if (score >= 80)
  System.out.print("B");
else if (score >= 70)
  System.out.print("C");
else if (score >= 60)
  System.out.print("D");
else
  System.out.print("F");
```

(b)

FIGURE 3.3 A preferred format for multiple alternatives is shown in (b) using a multi-way `if-else` statement.

The execution of this `if` statement proceeds as shown in Figure 3.4. The first condition `(score >= 90)` is tested. If it is `true`, the grade is `A`. If it is `false`, the second condition `(score >= 80)` is tested. If the second condition is `true`, the grade is `B`. If that condition is `false`, the third condition and the rest of the conditions (if necessary) are tested until a condition is met or all of the conditions prove to be `false`. If all of the conditions are `false`, the grade is `F`. Note a condition is tested only when all of the conditions that come before it are `false`.

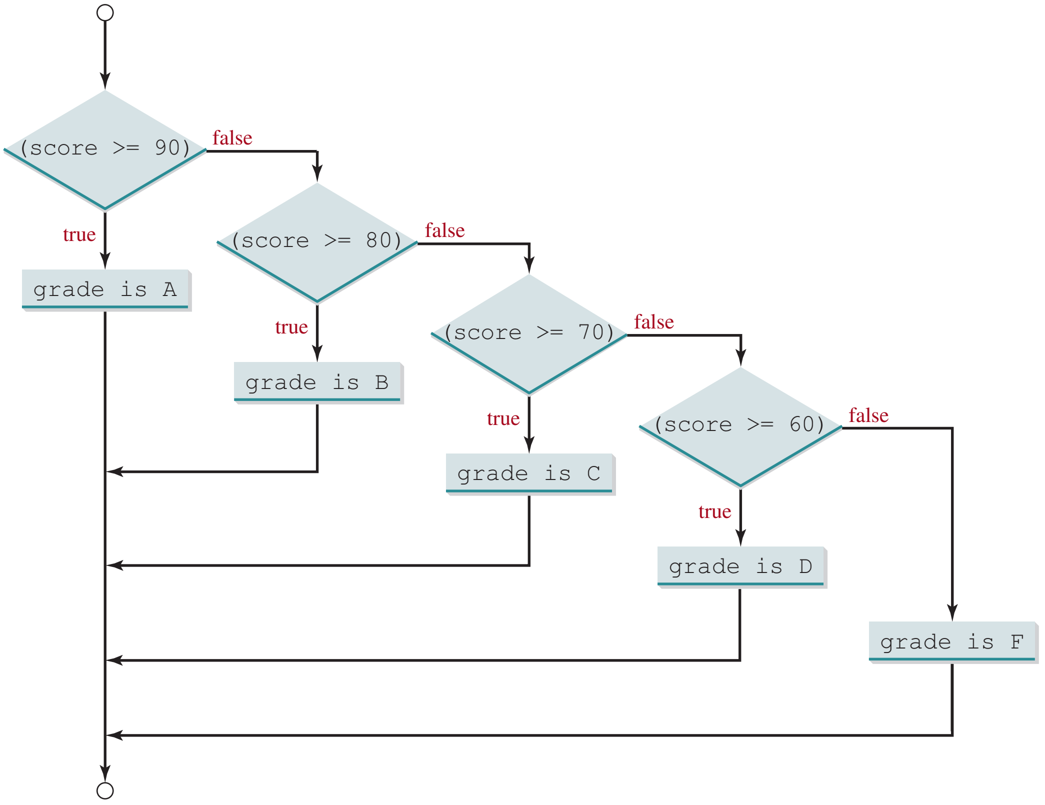

(score >= 90)
false
true
grade is A
(score >= 80)
false
true
grade is B
(score >= 70)
false
true
grade is C
(score >= 60)
false
true
grade is D
grade is F

FIGURE 3.4 You can use a multi-way `if-else` statement to assign a grade.

The `if` statement in Figure 3.3a is equivalent to the `if` statement in Figure 3.3b. In fact, Figure 3.3b is the preferred coding style for multiple alternative `if` statements. This style, called *multi-way* `if-else` *statements*, avoids deep indentation and makes the program easy to read.

multi-way `if` statement

Check Point

3.5.1 Suppose `x = 3` and `y = 2`; show the output, if any, of the following code. What is the output if `x = 3` and `y = 4`? What is the output if `x = 2` and `y = 2`? Draw a flowchart of the code.

```
if (x > 2) {
  if (y > 2) {
    z = x + y;
    System.out.println("z is " + z);
  }
}
else
  System.out.println("x is " + x);
```

3.5.2 Suppose `x = 2` and `y = 3`. Show the output, if any, of the following code. What is the output if `x = 3` and `y = 2`? What is the output if `x = 3` and `y = 3`?

```
if (x > 2)
  if (y > 2) {
    int z = x + y;
      System.out.println("z is " + z);
  }
else
  System.out.println("x is " + x);
```

3.5.3 What is wrong in the following code?

```
if (score >= 60)
  System.out.println("D");
else if (score >= 70)
  System.out.println("C");
else if (score >= 80)
  System.out.println("B");
else if (score >= 90)
  System.out.println("A");
else
  System.out.println("F");
```

3.6 Common Errors and Pitfalls

Key Point

Forgetting necessary braces, ending an `if` *statement in the wrong place, mistaking* `==` *for* `=`*, and dangling* `else` *clauses are common errors in selection statements. Duplicated statements in* `if-else` *statements and testing equality of double values are common pitfalls.*

The following errors are common among new programmers.

Common Error 1: Forgetting Necessary Braces

The braces can be omitted if the block contains a single statement. However, forgetting the braces when they are needed for grouping multiple statements is a common programming error. If you modify the code by adding new statements in an `if` statement without braces, you will have to insert the braces. For example, the following code in (a) is wrong. It should be written with braces to group multiple statements, as shown in (b).

```
if (radius >= 0)
  area = radius * radius * PI;
  System.out.println("The area "
    + " is " + area);
```

(a) Wrong

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area "
    + " is " + area);
}
```

(b) Correct

In (a), the console output statement is not part of the `if` statement. It is the same as the following code:

```
if (radius >= 0)
  area = radius * radius * PI;

System.out.println(“The area "
  + “ is " + area);
```

Regardless of the condition in the `if` statement, the console output statement is always executed.

Common Error 2: Wrong Semicolon at the `if` Line

Adding a semicolon at the end of an `if` line, as shown in (a) below, is a common mistake.

Logic error

```
if (radius >= 0);
{
  area = radius * radius * PI;
  System.out.println("The area "
    + " is " + area);
}
```

(a)

Equivalent

Empty block

```
if (radius >= 0) { };
{
  area = radius * radius * PI;
  System.out.println("The area "
    + " is " + area);
}
```

(b)

This mistake is hard to find, because it is neither a compile error nor a runtime error; it is a logic error. The code in (a) is equivalent to that in (b) with an empty block.

This error often occurs when you use the next-line block style. Using the end-of-line block style can help prevent this error.

Common Error 3: Redundant Testing of Boolean Values

To test whether a `boolean` variable is `true` or `false` in a test condition, it is redundant to use the equality testing operator like the code in (a):

```
if (even == true)
  System.out.println(
    "It is even.");
```

(a)

Equivalent

This is better

```
if (even)
  System.out.println(
    "It is even.");
```

(b)

Instead, it is better to test the `boolean` variable directly, as shown in (b). Another good reason for doing this is to avoid errors that are difficult to detect. Using the `=` operator instead of the `==` operator to compare the equality of two items in a test condition is a common error. It could lead to the following erroneous statement:

```
if (even = true)
  System.out.println("It is even.");
```

This statement does not have compile errors. It assigns `true` to `even`, so `even` is always `true`.

Common Error 4: Dangling `else` Ambiguity

dangling else ambiguity

The code in (a) below has two `if` clauses and one `else` clause. Which `if` clause is matched by the `else` clause? The indentation indicates that the `else` clause matches the first `if` clause. However, the `else` clause actually matches the second `if` clause. This situation is known as the *dangling else ambiguity*. The `else` clause always matches the most recent unmatched `if` clause in the same block. Therefore, the statement in (a) is equivalent to the code in (b).

```
int i = 1, j = 2, k = 3;

if (i > j)
  if (i > k)
     System.out.println("A");
else
     System.out.println("B");
```

(a)

Equivalent

This is better with correct indentation

```
int i = 1, j = 2, k = 3;

if (i > j)
  if (i > k)
    System.out.println("A");
  else
    System.out.println("B");
```

(b)

Since `(i > j)` is false, nothing is displayed from the statements in (a) and (b). To force the `else` clause to match the first `if` clause, you must add a pair of braces:

```
int i = 1, j = 2, k = 3;

if (i > j) {
  if (i > k)
    System.out.println("A");
}
else
  System.out.println("B");
```

This statement displays `B`.

Common Error 5: Equality Test of Two Floating-Point Values

As discussed in Common Error 3 in Section 2.19, floating-point numbers have a limited precision and calculations; involving floating-point numbers can introduce round-off errors. Therefore, equality test of two floating-point values is not reliable. For example, you expect the following code to display `true`, but surprisingly, it displays `false`:

```
double x = 1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1;
System.out.println(x == 0.5);
```

Here, `x` is not exactly `0.5`, but is `0.5000000000000001`. You cannot reliably test equality of two floating-point values. However, you can compare whether they are close enough by testing whether the difference of the two numbers is less than some threshold. That is, two numbers x and y are very close if $|x - y| < \varepsilon$, for a very small value, ε. ε, a Greek letter pronounced "epsilon", is commonly used to denote a very small value. Normally, you set ε to 10^{-14} for comparing two values of the `double` type, and to 10^{-7} for comparing two values of the `float` type. For example, the following code

```
final double EPSILON = 1E-14;
double x = 1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1;
if (Math.abs(x - 0.5) < EPSILON)
  System.out.println(x + " is approximately 0.5");
```

will display

```
0.5000000000000001 is approximately 0.5.
```

The `Math.abs(a)` method can be used to return the absolute value of `a`.

Common Pitfall 1: Simplifying Boolean Variable Assignment

Often, new programmers write the code that assigns a test condition to a `boolean` variable like the code in (a):

```
if (number % 2 == 0)
  even = true;
else
  even = false;
```

(a)

Equivalent

This is better

```
boolean even
  = number % 2 == 0;
```

(b)

This is not an error, but it should be better written as shown in (b).

Common Pitfall 2: Avoiding Duplicate Code in Different Cases

Often, new programmers write the duplicate code in different cases that should be combined in one place. For example, the highlighted code in the following statement is duplicated:

```
if (inState) {
  tuition = 5000;
  System.out.println("The tuition is " + tuition);
}
else {
  tuition = 15000;
  System.out.println("The tuition is " + tuition);
}
```

This is not an error, but it should be better written as follows:

```
if (inState) {
  tuition = 5000;
}
else {
  tuition = 15000;
}
System.out.println("The tuition is " + tuition);
```

The new code removes the duplication and makes the code easy to maintain, because you only need to change in one place if the print statement is modified.

Check Point

3.6.1 Which of the following statements are equivalent? Which ones are correctly indented?

```
if (i > 0) if
(j > 0)
x = 0; else
if (k > 0) y = 0;
else z = 0;
```

(a)

```
if (i > 0) {
  if (j > 0)
    x = 0;
  else if (k > 0)
    y = 0;
}
else
  z = 0;
```

(b)

```
if (i > 0)
  if (j > 0)
    x = 0;
   else if (k > 0)
    y = 0;
   else
    z = 0;
```

(c)

```
if (i > 0)
  if (j > 0)
    x = 0;
  else if (k > 0)
    y = 0;
else
  z = 0;
```

(d)

3.6.2 Rewrite the following statement using a Boolean expression:

```
if (count % 10 == 0)
  newLine = true;
else
  newLine = false;
```

3.6.3 Are the following statements correct? Which one is better?

```
if (age < 16)
  System.out.println
    ("Cannot get a driver's license");
if (age >= 16)
  System.out.println
    ("Can get a driver's license");
```

(a)

```
if (age < 16)
  System.out.println
    ("Cannot get a driver's license");
else
  System.out.println
    ("Can get a driver's license");
```

(b)

3.6.4 What is the output of the following code if `number` is `14`, `15`, or `30`?

```
if (number % 2 == 0)
  System.out.println
    (number +  " is even");
if (number % 5 == 0)
  System.out.println
    (number +  " is multiple of 5");
```

(a)

```
if (number % 2 == 0)
  System.out.println
    (number +  " is even");
else if (number % 5 == 0)
  System.out.println
    (number +  " is multiple of 5");
```

(b)

3.7 Generating Random Numbers

You can use `Math.random()` *to obtain a random double value between* `0.0` *and* `1.0`*, excluding* `1.0`.

Key Point

Suppose you want to develop a program for a first-grader to practice subtraction. The program randomly generates two single-digit integers, `number1` and `number2`, with `number1 >= number2`, and it displays to the student a question such as "What is 9 − 2?" After the student enters the answer, the program displays a message indicating whether it is correct.

VideoNote
Program subtraction quiz

The previous programs generate random numbers using `System.currentTimeMillis()`. A better approach is to use the `random()` method in the `Math` class. Invoking this method returns a random double value `d` such that $0.0 \le d < 1.0$. Thus, `(int)(Math.random() * 10)` returns a random single-digit integer (i.e., a number between `0` and `9`).

`random()` method

The program can work as follows:

1. Generate two single-digit integers into `number1` and `number2`.
2. If `number1 < number2`, swap `number1` with `number2`.
3. Prompt the student to answer, `"What is number1 – number2?"`
4. Check the student's answer and display whether the answer is correct.

The complete program is given in Listing 3.3.

Listing 3.3 `SubtractionQuiz.java`

random number

```
import java.util.Scanner;

public class SubtractionQuiz {
  public static void main(String[] args) {
    // 1. Generate two random single-digit integers
    int number1 = (int)(Math.random() * 10);
    int number2 = (int)(Math.random() * 10);

    // 2. If number1 < number2, swap number1 with number2
    if (number1 < number2) {
      int temp = number1;
```

get answer

check the answer

```
12       number1 = number2;
13       number2 = temp;
14     }
15
16     // 3. Prompt the student to answer "What is number1 - number2?"
17     System.out.print
18       ("What is " + number1 + " - " + number2 + "? ");
19     Scanner input = new Scanner(System.in);
20     int answer = input.nextInt();
21
22     // 4. Grade the answer and display the result
23     if (number1 - number2 == answer)
24       System.out.println("You are correct!");
25     else {
26       System.out.println("Your answer is wrong.");
27       System.out.println(number1 + " - " + number2 +
28         " should be " + (number1 - number2));
29     }
30   }
31 }
```

```
What is 6 - 6? 0 ↵Enter
You are correct!
```

```
What is 9 - 2? 5 ↵Enter
Your answer is wrong
9 - 2 is 7
```

line#	number1	number2	temp	answer	output
6	2				
7		9			
11			2		
12	9				
13		2			
20				5	
26					Your answer is wrong 9 - 2 should be 7

To swap two variables `number1` and `number2`, a temporary variable temp (line 11) is used to first hold the value in `number1`. The value in `number2` is assigned to `number1` (line 12), and the value in `temp` is assigned to `number2` (line 13).

Check Point

3.7.1 Which of the following is a possible output from invoking `Math.random()`?

`323.4`, `0.5`, `34`, `1.0`, `0.0`, `0.234`

3.7.2
- a. How do you generate a random integer `i` such that $0 \leq i < 20$?
- b. How do you generate a random integer `i` such that $10 \leq i < 20$?
- c. How do you generate a random integer `i` such that $10 \leq i \leq 50$?
- d. Write an expression that returns `0` or `1` randomly.

3.8 Case Study: Computing Body Mass Index

Key Point

You can use nested `if` *statements to write a program that interprets body mass index.*

Body mass index (BMI) is a measure of health based on height and weight. It can be calculated by taking your weight in kilograms and dividing it by the square of your height in meters. The interpretation of BMI for people 20 years or older is as follows:

BMI	Interpretation
$\text{BMI} < 18.5$	Underweight
$18.5 \le \text{BMI} < 25.0$	Normal
$25.0 \le \text{BMI} < 30.0$	Overweight
$30.0 \le \text{BMI}$	Obese

Write a program that prompts the user to enter a weight in pounds and height in inches and displays the BMI. Note that one pound is `0.45359237` kilograms, and one inch is `0.0254` meters. Listing 3.4 gives the program.

LISTING 3.4 ComputeAndInterpretBMI.java

```
import java.util.Scanner;

public class ComputeAndInterpretBMI {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter weight in pounds
    System.out.print("Enter weight in pounds: ");
    double weight = input.nextDouble();

    // Prompt the user to enter height in inches
    System.out.print("Enter height in inches: ");
    double height = input.nextDouble();

    final double KILOGRAMS_PER_POUND = 0.45359237; // Constant
    final double METERS_PER_INCH = 0.0254; // Constant

    // Compute BMI
    double weightInKilograms = weight * KILOGRAMS_PER_POUND;
    double heightInMeters = height * METERS_PER_INCH;
    double bmi = weightInKilograms /
      (heightInMeters * heightInMeters);

    // Display result
    System.out.println("BMI is " + bmi);
    if (bmi < 18.5)
      System.out.println("Underweight");
    else if (bmi < 25)
      System.out.println("Normal");
    else if (bmi < 30)
      System.out.println("Overweight");
    else
      System.out.println("Obese");
  }
}
```

input weight

input height

compute bmi

display output

```
Enter weight in pounds: 146 ↵Enter
Enter height in inches: 70 ↵Enter
BMI is 20.948603801493316
Normal
```

line#	weight	height	weightInKilograms	heightInMeters	bmi	output
9	146					
13		70				
19			66.22448602			
20				1.778		
21					20.9486	
25						BMI is 20.95
29						Normal

The constants **`KILOGRAMS_PER_POUND`** and **`METERS_PER_INCH`** are defined in lines 15–16. Using constants here makes programs easy to read.

You should test the input that covers all possible cases for BMI to ensure that the program works for all cases.

3.9 Case Study: Computing Taxes

Key Point

You can use nested `if` statements to write a program for computing taxes.

VideoNote

Use multi-way `if-else` statements

The U.S. federal personal income tax is calculated based on filing status and taxable income. There are four filing statuses: single filers, married filing jointly or qualified widow(er), married filing separately, and head of household. The tax rates vary every year. Table 3.2 shows the rates for 2009. If you are single with a taxable income of $10,000, for example, the first $8,350 is taxed at 10% and the other $1,650 is taxed at 15%, so your total tax is $1,082.50.

TABLE 3.2 2009 U.S. Federal Personal Tax Rates

Marginal Tax Rate	*Single*	*Married Filing Jointly or Qualifying Widow(er)*	*Married Filing Separately*	*Head of Household*
10%	$0–$8,350	$0–$16,700	$0–$8,350	$0–$11,950
15%	$8,351–$33,950	$16,701–$67,900	$8,351–$33,950	$11,951–$45,500
25%	$33,951–$82,250	$67,901–$137,050	$33,951–$68,525	$45,501–$117,450
28%	$82,251–$171,550	$137,051–$208,850	$68,526–$104,425	$117,451–$190,200
33%	$171,551–$372,950	$208,851–$372,950	$104,426–$186,475	$190,201–$372,950
35%	$372,951+	$372,951+	$186,476+	$372,951+

You are to write a program to compute personal income tax. Your program should prompt the user to enter the filing status and taxable income and compute the tax. Enter **`0`** for single filers, **`1`** for married filing jointly or qualified widow(er), **`2`** for married filing separately, and **`3`** for head of household.

Your program computes the tax for the taxable income based on the filing status. The filing status can be determined using `if` statements outlined as follows:

```
if (status == 0) {
  // Compute tax for single filers
}
else if (status == 1) {
  // Compute tax for married filing jointly or qualifying widow(er)
}
else if (status == 2) {
  // Compute tax for married filing separately
}
else if (status == 3) {
  // Compute tax for head of household
}
else  {
  // Display wrong status
}
```

For each filing status there are six tax rates. Each rate is applied to a certain amount of taxable income. For example, of a taxable income of $400,000 for single filers, $8,350 is taxed at 10%, (33,950 − 8,350) at 15%, (82,250 − 33,950) at 25%, (171,550 − 82,250) at 28%, (372,950 − 171,550) at 33%, and (400,000 − 372,950) at 35%.

Listing 3.5 gives the solution for computing taxes for single filers. The complete solution is left as an exercise.

LISTING 3.5 ComputeTax.java

```
import java.util.Scanner;

public class ComputeTax {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter filing status
    System.out.print("(0-single filer, 1-married jointly or " +
      "qualifying widow(er), 2-married separately, 3-head of " +
      "household) Enter the filing status: ");

    int status = input.nextInt();

    // Prompt the user to enter taxable income
    System.out.print("Enter the taxable income: ");
    double income = input.nextDouble();

    // Compute tax
    double tax = 0;

    if (status == 0) { // Compute tax for single filers
      if (income <= 8350)
        tax = income * 0.10;
      else if (income <= 33950)
        tax = 8350 * 0.10 + (income - 8350) * 0.15;
      else if (income <= 82250)
        tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
          (income - 33950) * 0.25;
      else if (income <= 171550)
        tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
          (82250 - 33950) * 0.25 + (income - 82250) * 0.28;
```

input status

input income

compute tax

```
33       else if (income <= 372950)
34         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
35           (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
36           (income - 171550) * 0.33;
37       else
38         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
39           (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
40           (372950 - 171550) * 0.33 + (income - 372950) * 0.35;
41     }
42     else if (status == 1) { // Left as an exercise
43       // Compute tax for married file jointly or qualifying widow(er)
44     }
45     else if (status == 2) { // Compute tax for married separately
46       // Left as an exercise in Programming Exercise 3.13
47     }
48     else if (status == 3) { // Compute tax for head of household
49       // Left as an exercise in Programming Exercise 3.13
50     }
51     else {
52       System.out.println("Error: invalid status");
53       System.exit(1);
54     }
55 
56     // Display the result
57     System.out.println("Tax is " + (int)(tax * 100) / 100.0);
58   }
59 }
```

exit program

display output

```
(0-single filer, 1-married jointly or qualifying widow(er),
2-married separately, 3-head of household)
Enter the filing status: 0 ↵Enter
Enter the taxable income: 400000 ↵Enter
Tax is 117683.5
```

line#	status	income	Tax	output
13	0			
17		400000		
20			0	
38			117683.5	
57				Tax is 117683.5

The program receives the filing status and taxable income. The multi-way **`if-else`** statements (lines 22, 42, 45, 48, and 51) check the filing status and compute the tax based on the filing status.

System.exit(status)

`System.exit(status)` (line 53) is defined in the **`System`** class. Invoking this method terminates the program. The status **`0`** indicates that the program is terminated normally. A nonzero status code indicates abnormal termination.

An initial value of **`0`** is assigned to **`tax`** (line 20). A compile error would occur if it had no initial value, because all of the other statements that assign values to **`tax`** are within the **`if`** statement. The compiler thinks these statements may not be executed, and therefore reports a compile error.

To test a program, you should provide the input that covers all cases. For this program, your input should cover all statuses (`0`, `1`, `2`, `3`). For each status, test the tax for each of the six brackets. Thus, there are a total of 24 cases.

test all cases

Tip

For all programs, you should write a small amount of code and test it before moving on to add more code. This is called *incremental development and testing*. This approach makes testing easier, because the errors are likely in the new code you just added.

incremental development and testing

3.9.1 Are the following two statements equivalent?

Check Point

```
if (income <= 10000)
  tax = income * 0.1;
else if (income <= 20000)
  tax = 1000 +
    (income - 10000) * 0.15;
```

```
if (income <= 10000)
  tax = income * 0.1;
else if (income > 10000 &&
         income <= 20000)
  tax = 1000 +
    (income - 10000) * 0.15;
```

3.10 Logical Operators

The logical operators `!`, `&&`, `||`, *and* `^` *can be used to create a compound Boolean expression.*

Key Point

Sometimes, whether a statement is executed is determined by a combination of several conditions. You can use logical operators to combine these conditions to form a compound Boolean expression. *Logical operators*, also known as *Boolean operators*, operate on Boolean values to create a new Boolean value. Table 3.3 lists the Boolean operators. Table 3.4 defines the not (`!`) operator, which negates `true` to `false` and `false` to `true`. Table 3.5 defines the and (`&&`) operator. The and (`&&`) of two Boolean operands is `true` if and only if both the operands are `true`. Table 3.6 defines the or (`||`) operator. The or (`||`) of two Boolean operands is `true` if at least one of the operands is `true`. Table 3.7 defines the exclusive or (`^`) operator. The exclusive or (`^`) of two Boolean operands is `true` if and only if the two operands have different Boolean values. Note `p1 ^ p2` is the same as `p1 != p2`.

TABLE 3.3 Boolean Operators

Operator	*Name*	*Description*
`!`	not	Logical negation
`&&`	and	Logical conjunction
`||`	or	Logical disjunction
`^`	exclusive or	Logical exclusion

TABLE 3.4 Truth Table for Operator `!`

`p`	`!p`	*Example (assume* `age = 24, weight = 140`*)*
`true`	`false`	`!(age > 18)` is `false`, because `(age > 18)` is `true`.
`false`	`true`	`!(weight == 150)` is `true`, because `(weight == 150)` is `false`.

TABLE 3.5 Truth Table for Operator `&&`

p_1	p_2	p_1 `&&` p_2	*Example (assume* `age = 24, weight = 140`*)*
`false`	`false`	`false`	
`false`	`true`	`false`	`(age > 28) && (weight <= 140)` is `false`, because `(age > 28)` is `false`.
`true`	`false`	`false`	
`true`	`true`	`true`	`(age > 18) && (weight >= 140)` is `true`, because `(age > 18)` and `(weight >= 140)` are both `true`.

TABLE 3.6 Truth Table for Operator `||`

p_1	p_2	p_1 `||` p_2	*Example (assume* `age = 24, weight = 140`*)*
`false`	`false`	`false`	`(age > 34) || (weight >= 150)` is `false`, because `(age > 34)` and `(weight >= 150)` are both `false`.
`false`	`true`	`true`	
`true`	`false`	`true`	`(age > 18) || (weight < 140)` is `true`, because `(age > 18)` is `true`.
`true`	`true`	`true`	

TABLE 3.7 Truth Table for Operator `^`

p_1	p_2	p_1 `^` p_2	*Example (assume* `age = 24, weight = 140`*)*
`false`	`false`	`false`	`(age > 34) ^ (weight > 140)` is `false`, because `(age > 34)` and `(weight > 140)` are both `false`.
`false`	`true`	`true`	`(age > 34) ^ (weight >= 140)` is `true`, because `(age > 34)` is `false` but `(weight >= 140)` is `true`.
`true`	`false`	`true`	
`true`	`true`	`false`	

Listing 3.6 gives a program that checks whether a number is divisible by **2** and **3**, by **2** or **3**, and by **2** or **3** but not both.

LISTING 3.6 TestBooleanOperators.java

import class

input

and

```java
import java.util.Scanner;

public class TestBooleanOperators {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Receive an input
    System.out.print("Enter an integer: ");
    int number = input.nextInt();

    if (number % 2 == 0 && number % 3 == 0)
      System.out.println(number + " is divisible by 2 and 3.");

15      if (number % 2 == 0 || number % 3 == 0)
16        System.out.println(number + " is divisible by 2 or 3.");
17
18      if (number % 2 == 0 ^ number % 3 == 0)
19        System.out.println(number +
20          " is divisible by 2 or 3, but not both.");
21    }
22  }
```

or

exclusive or

```
Enter an integer: 4 ↵Enter
4 is divisible by 2 or 3.
4 is divisible by 2 or 3, but not both.
```

```
Enter an integer: 18 ↵Enter
18 is divisible by 2 and 3.
18 is divisible by 2 or 3.
```

`(number % 2 == 0 && number % 3 == 0)` (line 12) checks whether the number is divisible by both `2` and `3`. `(number % 2 == 0 || number % 3 == 0)` (line 15) checks whether the number is divisible by `2` or by `3`. `(number % 2 == 0 ^ number % 3 == 0)` (line 18) checks whether the number is divisible by `2` or `3`, but not both.

Caution

In mathematics, the expression

```
28 <= numberOfDaysInAMonth <= 31
```

is correct. However, it is incorrect in Java, because `28 <= numberOfDaysInAMonth` is evaluated to a `boolean` value, which cannot be compared with `31`. Here, two operands (a `boolean` value and a numeric value) are *incompatible*. The correct expression in Java is

incompatible operands

```
28 <= numberOfDaysInAMonth && numberOfDaysInAMonth <= 31
```

Note

De Morgan's law, named after Indian-born British mathematician and logician Augustus De Morgan (1806–1871), can be used to simplify Boolean expressions. The law states the following:

De Morgan's law

`!(condition1 && condition2)` is the same as
`!condition1 || !condition2`
`!(condition1 || condition2)` is the same as
`!condition1 && !condition2`

For example,

```
!(number % 2 == 0 && number % 3 == 0)
```

can be simplified using an equivalent expression:

```
number % 2 != 0 || number % 3 != 0
```

As another example,

```
!(number == 2 || number == 3)
```

is better written as

```
number != 2 && number != 3
```

If one of the operands of an `&&` operator is `false`, the expression is `false`; if one of the operands of an `||` operator is `true`, the expression is `true`. Java uses these properties to improve the performance of these operators. When evaluating `p1 && p2`, Java first evaluates `p1` then, if `p1` is `true`, evaluates `p2`; if `p1` is `false`, it does not evaluate `p2`. When evaluating `p1 || p2`, Java first evaluates `p1` then, if `p1` is `false`, evaluates `p2`; if `p1` is `true`, it does not evaluate `p2`. In programming language terminology, `&&` and `||` are known as the *short-circuit* or *lazy operators*. Java also provides the `&` and `|` operators, which are covered in Supplement III.C for advanced readers.

short-circuit operator
lazy operator

Check Point

3.10.1 Assuming that `x` is `1`, show the result of the following Boolean expressions:

```
(true) && (3 > 4)
!(x > 0) && (x > 0)
(x > 0) || (x < 0)
(x != 0) || (x == 0)
(x >= 0) || (x < 0)
(x != 1) == !(x == 1)
```

3.10.2 (a) Write a Boolean expression that evaluates to `true` if a number stored in variable `num` is between `1` and `100`. (b) Write a Boolean expression that evaluates to `true` if a number stored in variable `num` is between `1` and `100` or the number is negative.

3.10.3 (a) Write a Boolean expression for $|x - 5| < 4.5$. (b) Write a Boolean expression for $|x - 5| > 4.5$.

3.10.4 Assume `x` and `y` are `int` type. Which of the following are legal Java expressions?

```
x > y > 0
x = y && y
x /= y
x or y
x and y
(x != 0) || (x = 0)
```

3.10.5 Are the following two expressions the same?

(a) `x % 2 == 0 && x % 3 == 0`

(b) `x % 6 == 0`

3.10.6 What is the value of the expression `x >= 50 && x <= 100` if `x` is `45`, `67`, or `101`?

3.10.7 Suppose, when you run the following program, you enter the input `2 3 6` from the console. What is the output?

```
public class Test {
  public static void main(String[] args) {
    java.util.Scanner input = new java.util.Scanner(System.in);
    double x = input.nextDouble();
    double y = input.nextDouble();
    double z = input.nextDouble();

    System.out.println("(x < y && y < z) is " + (x < y && y < z));
    System.out.println("(x < y || y < z) is " + (x < y || y < z));
    System.out.println("!(x < y) is " + !(x < y));
    System.out.println("(x + y < z) is " + (x + y < z));
    System.out.println("(x + y > z) is " + (x + y > z));
  }
}
```

3.10.8 Write a Boolean expression that evaluates to `true` if `age` is greater than `13` and less than `18`.

3.10.9 Write a Boolean expression that evaluates to `true` if `weight` is greater than `50` pounds or height is greater than `60` inches.

3.10.10 Write a Boolean expression that evaluates to `true` if `weight` is greater than `50` pounds and height is greater than `60` inches.

3.10.11 Write a Boolean expression that evaluates to `true` if either `weight` is greater than `50` pounds or height is greater than `60` inches, but not both.

3.11 Case Study: Determining Leap Year

Key Point

A year is a leap year if it is divisible by `4` *but not by* `100`*, or if it is divisible by* `400`*.*

A leap year has 366 days. The February of a leap year has 29 days. You can use the following Boolean expressions to check whether a year is a leap year:

```
// A leap year is divisible by 4
boolean isLeapYear = (year % 4 == 0);

// A leap year is divisible by 4 but not by 100
isLeapYear = isLeapYear && (year % 100 != 0);

// A leap year is divisible by 4 but not by 100 or divisible by 400
isLeapYear = isLeapYear || (year % 400 == 0);
```

Or you can combine all these expressions into one as follows:

```
isLeapYear = (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
```

Listing 3.7 gives the program that lets the user enter a year and checks whether it is a leap year.

LISTING 3.7 LeapYear.java

```
import java.util.Scanner;

public class LeapYear {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a year: ");
    int year = input.nextInt();

    // Check if the year is a leap year
    boolean isLeapYear =
      (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);

    // Display the result
    System.out.println(year + " is a leap year? " + isLeapYear);
  }
}
```

input

leap year?

display result

```
Enter a year: 2008 ↵Enter
2008 is a leap year? true
```

```
Enter a year: 1900 ↵Enter
1900 is a leap year? false
```

```
Enter a year: 2002 ↵Enter
2002 is a leap year? false
```

3.11.1 How many days in the February of a leap year? Which of the following is a leap year? 500, 1000, 2000, 2016, and 2020?

3.12 Case Study: Lottery

The lottery program involves generating random numbers, comparing digits, and using Boolean operators.

Suppose you want to develop a program to play lottery. The program randomly generates a lottery of a two-digit number, prompts the user to enter a two-digit number, and determines whether the user wins according to the following rules:

1. If the user input matches the lottery number in the exact order, the award is $10,000.
2. If all digits in the user input match all digits in the lottery number, the award is $3,000.
3. If one digit in the user input matches a digit in the lottery number, the award is $1,000.

Note the digits of a two-digit number may be `0`. If a number is less than `10`, we assume that the number is preceded by a `0` to form a two-digit number. For example, number `8` is treated as `08`, and number `0` is treated as `00` in the program. Listing 3.8 gives the complete program.

Listing 3.8 Lottery.java

generate a lottery number

enter a guess

exact match?

match all digits?

match one digit?

```
import java.util.Scanner;

public class Lottery {
  public static void main(String[] args) {
    // Generate a lottery number
    int lottery = (int)(Math.random() * 100);

    // Prompt the user to enter a guess
    Scanner input = new Scanner(System.in);
    System.out.print("Enter your lottery pick (two digits): ");
    int guess = input.nextInt();

    // Get digits from lottery
    int lotteryDigit1 = lottery / 10;
    int lotteryDigit2 = lottery % 10;

    // Get digits from guess
    int guessDigit1 = guess / 10;
    int guessDigit2 = guess % 10;

    System.out.println("The lottery number is " + lottery);

    // Check the guess
    if (guess == lottery)
      System.out.println("Exact match: you win $10,000");
    else if (guessDigit2 == lotteryDigit1
           && guessDigit1 == lotteryDigit2)
      System.out.println("Match all digits: you win $3,000");
    else if (guessDigit1 == lotteryDigit1
          || guessDigit1 == lotteryDigit2
          || guessDigit2 == lotteryDigit1
          || guessDigit2 == lotteryDigit2)
33         System.out.println("Match one digit: you win $1,000");
34       else
35         System.out.println("Sorry, no match");
36     }
37   }
```

```
Enter your lottery pick (two digits): 15 ↵Enter
The lottery number is 15
Exact match: you win $10,000
```

```
Enter your lottery pick (two digits): 45 ↵Enter
The lottery number is 54
Match all digits: you win $3,000
```

```
Enter your lottery pick: 23 ↵Enter
The lottery number is 34
Match one digit: you win $1,000
```

```
Enter your lottery pick: 23 ↵Enter
The lottery number is 14
Sorry: no match
```

line# / variable	6	11	14	15	18	19	33
`lottery`	34						
`guess`		23					
`lotteryDigit1`			3				
`lotteryDigit2`				4			
`guessDigit1`					2		
`guessDigit2`						3	
`Output`							Match one digit: you win $1,000

The program generates a lottery using the `random()` method (line 6) and prompts the user to enter a guess (line 11). Note `guess % 10` obtains the last digit from `guess` and `guess /10` obtains the first digit from `guess`, since `guess` is a two-digit number (lines 18 and 19).

The program checks the guess against the lottery number in this order:

1. First, check whether the guess matches the lottery exactly (line 24).
2. If not, check whether the reversal of the guess matches the lottery (lines 26 and 27).
3. If not, check whether one digit is in the lottery (lines 29–32).
4. If not, nothing matches and display `"Sorry, no match"` (lines 34 and 35).

Check Point

3.12.1 What happens if you enter an integer as `05`?

3.13 `switch` Statements

Key Point

A `switch` *statement executes statements based on the value of a variable or an expression.*

The `if` statement in Listing 3.5, ComputeTax.java, makes selections based on a single `true` or `false` condition. There are four cases for computing taxes, which depend on the value of `status`. To fully account for all the cases, nested `if` statements were used. Overuse of nested `if` statements makes a program difficult to read. Java provides a `switch` statement to simplify coding for multiple conditions. You can write the following `switch` statement to replace the nested `if` statement in Listing 3.5:

```
switch (status) {
  case 0:  compute tax for single filers;
           break;
  case 1:  compute tax for married jointly or qualifying widow(er);
           break;
  case 2:  compute tax for married filing separately;
           break;
  case 3:  compute tax for head of household;
           break;
  default: System.out.println("Error: invalid status");
           System.exit(1);
}
```

The flowchart of the preceding `switch` statement is shown in Figure 3.5.

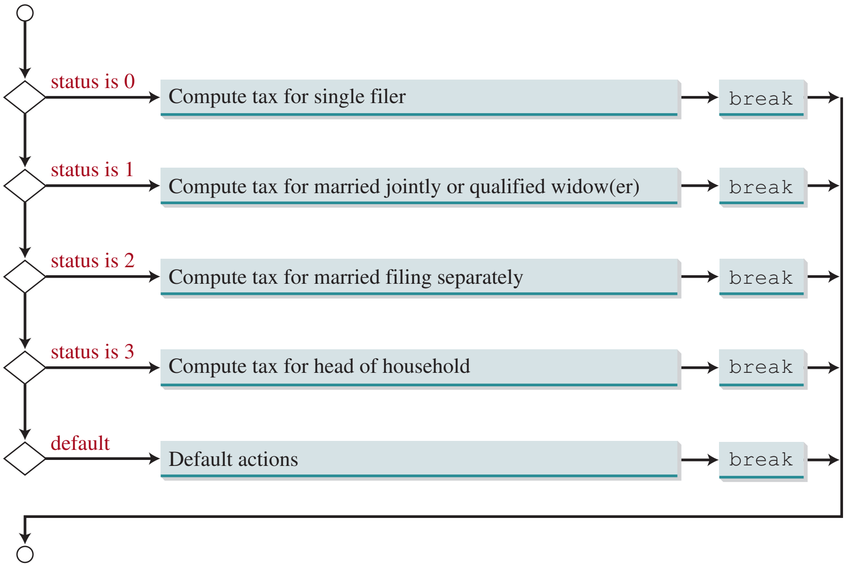

status is 0
Compute tax for single filer
break
status is 1
Compute tax for married jointly or qualified widow(er)
break
status is 2
Compute tax for married filing separately
break
status is 3
Compute tax for head of household
break
default
Default actions
break

FIGURE 3.5 The `switch` statement checks all cases and executes the statements in the matched case.

This statement checks to see whether the status matches the value `0`, `1`, `2`, or `3`, in that order. If matched, the corresponding tax is computed; if not matched, a message is displayed. Here is the full syntax for the `switch` statement:

switch statement

```
switch (switch-expression) {
  case value1: statement(s)1;
               break;
  case value2: statement(s)2;
               break;
  ...
  case valueN: statement(s)N;
               break;
  default:     statement(s)-for-default;
}
```

The `switch` statement observes the following rules:

- The `switch-expression` must yield a value of `char`, `byte`, `short`, `int`, or `String` type and must always be enclosed in parentheses. (The `char` and `String` types will be introduced in Chapter 4.)
- The `value1`, ..., and `valueN` must have the same data type as the value of the `switch-expression`. Note that `value1`, ..., and `valueN` are constant expressions, meaning they cannot contain variables, such as `1 + x`.
- When the value in a `case` statement matches the value of the `switch-expression`, the statements *starting from this case* are executed until either a `break` statement or the end of the `switch` statement is reached.
- The `default` case, which is optional, can be used to perform actions when none of the specified cases matches the `switch-expression`.
- The keyword `break` is optional. The `break` statement immediately ends the `switch` statement.

Caution

Do not forget to use a `break` statement when one is needed. Once a case is matched, the statements starting from the matched case are executed until a `break` statement or the end of the `switch` statement is reached. This is referred to as *fall-through behavior*. For example, the following code displays `Weekday` for days `1`–`5` and `Weekend` for day `0` and day `6`.

without break

fall-through behavior

```
switch (day) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5: System.out.println(“Weekday"); break;
  case 0:
  case 6: System.out.println(“Weekend");
}
```

Tip

To avoid programming errors and improve code maintainability, it is a good idea to put a comment in a case clause if `break` is purposely omitted.

Now let us write a program to find out the Chinese Zodiac sign for a given year. The Chinese Zodiac is based on a 12-year cycle, with each year represented by an animal—monkey, rooster, dog, pig, rat, ox, tiger, rabbit, dragon, snake, horse, or sheep—in this cycle, as shown in Figure 3.6.

Note `year % 12` determines the Zodiac sign. 1900 is the year of the rat because `1900 % 12` is `4`. Listing 3.9 gives a program that prompts the user to enter a year and displays the animal for the year.

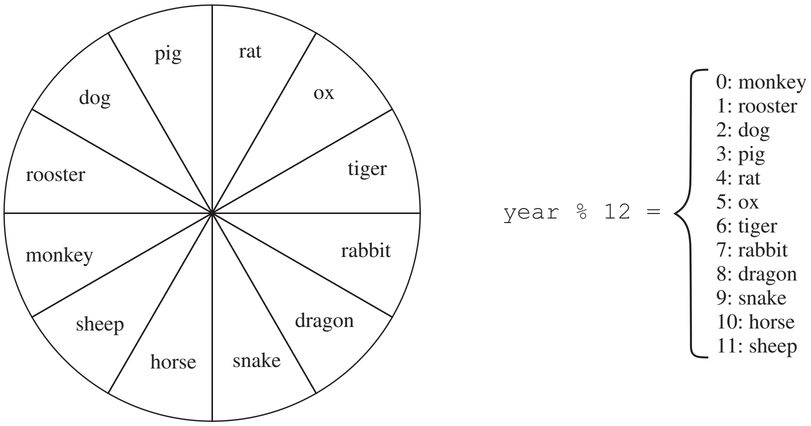

pig
rat
dog
ox
rooster
tiger
monkey
rabbit
sheep
dragon
horse
snake
year % 12 =
0: monkey
1: rooster
2: dog
3: pig
4: rat
5: ox
6: tiger
7: rabbit
8: dragon
9: snake
10: horse
11: sheep

FIGURE 3.6 The Chinese Zodiac is based on a 12-year cycle.

LISTING 3.9 ChineseZodiac.java

enter year

determine Zodiac sign

```java
import java.util.Scanner;

public class ChineseZodiac {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    System.out.print("Enter a year: ");
    int year = input.nextInt();

    switch (year % 12) {
      case 0: System.out.println("monkey"); break;
      case 1: System.out.println("rooster"); break;
      case 2: System.out.println("dog"); break;
      case 3: System.out.println("pig"); break;
      case 4: System.out.println("rat"); break;
      case 5: System.out.println("ox"); break;
      case 6: System.out.println("tiger"); break;
      case 7: System.out.println("rabbit"); break;
      case 8: System.out.println("dragon"); break;
      case 9: System.out.println("snake"); break;
      case 10: System.out.println("horse"); break;
      case 11: System.out.println("sheep");
    }
  }
}
```

```
Enter a year: 1963 ↵Enter
rabbit
```

```
Enter a year: 1877 ↵Enter
ox
```

Check Point

3.13.1 What data types are required for a `switch` variable? If the keyword `break` is not used after a case is processed, what is the next statement to be executed? Can you convert a `switch` statement to an equivalent `if` statement, or vice versa? What are the advantages of using a `switch` statement?

3.13.2 What is `y` after the following `switch` statement is executed? Rewrite the code using an `if-else` statement.

```
x = 3; y = 3;
switch (x +  3) {
  case 6: y = 1;
  default: y += 1;
}
```

3.13.3 What is `x` after the following `if-else` statement is executed? Use a `switch` statement to rewrite it and draw the flowchart for the new `switch` statement.

```
int x = 1, a = 3;
if (a == 1)
  x += 5;
else if (a == 2)
  x += 10;
else if (a == 3)
  x += 16;
else if (a == 4)
  x += 34;
```

3.13.4 Write a `switch` statement that displays Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, if `day` is `0`, `1`, `2`, `3`, `4`, `5`, `6`, respectively.

3.13.5 Rewrite Listing 3.9 using an `if-else` statement.

3.14 Conditional Operators

A conditional operator evaluates an expression based on a condition.

Key Point

You might want to assign a value to a variable that is restricted by certain conditions. For example, the following statement assigns `1` to `y` if `x` is greater than `0` and `-1` to `y` if `x` is less than or equal to `0`:

```
if (x > 0)
  y = 1;
else
  y = -1;
```

conditional operator

Alternatively, as in the following example, you can use a *conditional operator* to achieve the same result.

```
y = (x > 0)? 1: -1;
```

ternary operator

The symbols`?` and`:` appearing together is called a conditional operator (also known as a *ternary operator* because it uses three operands. It is the only ternary operator in Java. The conditional operator is in a completely different style, with no explicit `if` in the statement. The syntax to use the operator is as follows:

```
boolean-expression? expression1: expression2
```

The result of this expression is `expression1` if `boolean-expression` is true; otherwise the result is `expression2`.

Suppose you want to assign the larger number of variable `num1` and `num2` to `max`. You can simply write a statement using the conditional operator:

```
max = (num1 > num2)? num1: num2;
```

For another example, the following statement displays the message "num is even" if `num` is even, and otherwise displays "num is odd."

```
System.out.println((num % 2 == 0)? "num is even": "num is odd");
```

As you can see from these examples, the conditional operator enables you to write short and concise code.

Conditional expressions can be embedded. For example, the following code assigns `1`, `0`, or `-1` to status if `n1 > n1, n1 == n2`, or `n1 < n2`:

```
status = n1 > n2? 1: (n1 == n2? 0: -1);
```

Check Point

3.14.1 Suppose when you run the following program, you enter the input `2 3 6` from the console. What is the output?

```
public class Test {
  public static void main(String[] args) {
    java.util.Scanner input = new java.util.Scanner(System.in);
    double x = input.nextDouble();
    double y = input.nextDouble();
    double z = input.nextDouble();

    System.out.println((x < y && y < z)? "sorted": "not sorted");
  }
}
```

3.14.2 Rewrite the following `if` statements using the conditional operator.

```
if (ages >= 16)
  ticketPrice = 20;
else
  ticketPrice = 10;
```

3.14.3 Rewrite the following codes using `if-else` statements.

a. `score = (x > 10)? 3 * scale: 4 * scale;`
b. `tax = (income > 10000)? income * 0.2: income * 0.17 + 1000;`
c. `System.out.println((number % 3 == 0)? i: j);`

3.14.4 Write an expression using a conditional operator that returns randomly `-1` or `1`.

3.15 Operator Precedence and Associativity

Key Point

Operator precedence and associativity determine the order in which operators are evaluated.

Section 2.11 introduced operator precedence involving arithmetic operators. This section discusses operator precedence in more detail. Suppose you have this expression:

```
3 + 4 * 4 > 5 * (4 + 3) – 1 && (4 – 3 > 5)
```

What is its value? What is the execution order of the operators?

The expression within parentheses is evaluated first. (Parentheses can be nested, in which case the expression within the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule.

operator precedence

The precedence rule defines precedence for operators, as shown in Table 3.8, which contains the operators you have learned so far. Operators are listed in decreasing order of precedence from top to bottom. The logical operators have lower precedence than the relational operators, and the relational operators have lower precedence than the arithmetic operators. Operators with the same precedence appear in the same group. (See Appendix C, *Operator Precedence Chart*, for a complete list of Java operators and their precedence.)

TABLE 3.8 Operator Precedence Chart

Precedence	*Operator*
↓	`var++` and `var--` (Postfix)
	`+`, `-` (Unary plus and minus), `++var` and `--var` (Prefix)
	(type) (Casting)
	`!`(Not)
	`*`, `/`, `%` (Multiplication, division, and remainder)
	`+`, `-` (Binary addition and subtraction)
	`<`, `<=`, `>`, `>=` (Relational)
	`==`, `!=` (Equality)
	`^` (Exclusive OR)
	`&&` (AND)
	`||` (OR)
	`?:` (Ternary operator)
	`=`, `+=`, `-=`, `*=`, `/=`, `%=` (Assignment operators)

If operators with the same precedence are next to each other, their *associativity* determines the order of evaluation. All binary operators except assignment operators are *left associative*. For example, since `+` and `-` are of the same precedence and are left associative, the expression

operator associativity

$$\texttt{a - b + c - d}\ \overset{\text{is equivalent to}}{=\!=\!=\!=}\ \texttt{((a - b) + c) - d}$$

Assignment operators are *right associative*. Therefore, the expression

$$\texttt{a = b += c = 5}\ \overset{\text{is equivalent to}}{=\!=\!=\!=}\ \texttt{a = (b += (c = 5))}$$

Suppose `a`, `b`, and `c` are `1` before the assignment; after the whole expression is evaluated, `a` becomes `6`, `b` becomes `6`, and `c` becomes `5`. Note that left associativity for the assignment operator would not make sense.

Note

Java has its own way to evaluate an expression internally. The result of a Java evaluation is the same as that of its corresponding arithmetic evaluation. Advanced readers may refer to Supplement III.B for more discussions on how an expression is evaluated in Java *behind the scenes*.

behind the scenes

Check Point

3.15.1 List the precedence order of the Boolean operators. Evaluate the following expressions:

```
true || true && false
true && true || false
```

3.15.2 True or false? All the binary operators except `=` are left associative.

3.15.3 Evaluate the following expressions:

```
2 * 2 - 3 > 2 && 4 - 2 > 5
2 * 2 - 3 > 2 || 4 - 2 > 5
```

3.15.4 Is `(x > 0 && x < 10)` the same as `((x > 0) && (x < 10))`?

Is `(x > 0 || x < 10)` the same as `((x > 0) || (x < 10))`?

Is `(x > 0 || x < 10 && y < 0)` the same as `(x > 0 || (x < 10 && y < 0))`?

3.16 Debugging

Key Point

Debugging is the process of finding and fixing errors in a program.

As mentioned in Section 1.10, syntax errors are easy to find and easy to correct because the compiler gives indications as to where the errors came from and why they are there. Runtime errors are not difficult to find either, because the Java interpreter displays them on the console when the program aborts. Finding logic errors, on the other hand, can be very challenging.

bugs
hand-traces
debugging

Logic errors are called *bugs*. The process of finding and correcting errors is called *debugging*. A common approach to debugging is to use a combination of methods to help pinpoint the part of the program where the bug is located. You can *hand-trace* the program (i.e., catch errors by reading the program), or you can insert print statements in order to show the values of the variables or the execution flow of the program. These approaches might work for debugging a short, simple program, but for a large, complex program, the most effective approach is to use a debugger utility.

JDK includes a command-line debugger, jdb, which is invoked with a class name. jdb is itself a Java program, running its own copy of Java interpreter. All the Java IDE tools, such as Eclipse and NetBeans, include integrated debuggers. The debugger utilities let you follow the execution of a program. They vary from one system to another, but they all support most of the following helpful features.

- *Executing a single statement at a time:* The debugger allows you to execute one statement at a time so that you can see the effect of each statement.
- *Tracing into or stepping over a method:* If a method is being executed, you can ask the debugger to enter the method and execute one statement at a time in the method, or you can ask it to step over the entire method. You should step over the entire method if you know that the method works. For example, always step over system-supplied methods, such as `System.out.println`.
- **Setting breakpoints:** You can also set a breakpoint at a specific statement. Your program pauses when it reaches a breakpoint. You can set as many breakpoints as you want. Breakpoints are particularly useful when you know where your programming error starts. You can set a breakpoint at that statement, and have the program execute until it reaches the breakpoint.
- **Displaying variables:** The debugger lets you select several variables and display their values. As you trace through a program, the content of a variable is continuously updated.
- **Displaying call stacks:** The debugger lets you trace all of the method calls. This feature is helpful when you need to see a large picture of the program-execution flow.
- **Modifying variables:** Some debuggers enable you to modify the value of a variable when debugging. This is convenient when you want to test a program with different samples, but do not want to leave the debugger.

debugging in IDE

Tip

If you use an IDE such as Eclipse or NetBeans, please refer to *Learning Java Effectively with Eclipse/NetBeans* in Supplements II.C and II.E on the Companion Website. The supplement shows you how to use a debugger to trace programs, and how debugging can help in learning Java effectively.

Key Terms

boolean data type, 78
Boolean expression, 78
Boolean value, 78
conditional operator, 105
dangling else ambiguity, 87
debugging, 108
fall-through behavior, 103
flowchart, 80
lazy operator, 98
operator associativity, 107
operator precedence, 106
selection statement, 78
short-circuit operator, 98

Chapter Summary

1. A `boolean`-type variable can store a `true` or `false` value.

2. The relational operators (`<`, `<=`, `==`, `!=`, `>`, and `>=`) yield a Boolean value.

3. *Selection statements* are used for programming with alternative courses of actions. There are several types of selection statements: one-way `if` statements, two-way `if-else` statements, nested `if` statements, multi-way `if-else` statements, `switch` statements, and conditional operators.

4. The various `if` statements all make control decisions based on a *Boolean expression*. Based on the `true` or `false` evaluation of the expression, these statements take one of the two possible courses.

5. The Boolean operators `&&`, `||`, `!`, and `^` operate with Boolean values and variables.

6. When evaluating `p1 && p2`, Java first evaluates `p1` then evaluates `p2` if `p1` is `true`; if `p1` is `false`, it does not evaluate `p2`. When evaluating `p1 || p2`, Java first evaluates `p1` then evaluates `p2` if `p1` is `false`; if `p1` is `true`, it does not evaluate `p2`. Therefore, `&&` is referred to as the *short-circuit* or *lazy AND operator*, and `||` is referred to as the *short-circuit* or *lazy OR operator*.

7. The `switch` statement makes control decisions based on a switch expression of type `char`, `byte`, `short`, `int`, or `String`.

8. The keyword `break` is optional in a `switch` statement, but it is normally used at the end of each case in order to skip the remainder of the `switch` statement. If the `break` statement is not present, the next `case` statement will be executed.

9. The operators in expressions are evaluated in the order determined by the rules of parentheses, *operator precedence*, and *operator associativity*.

10. Parentheses can be used to force the order of evaluation to occur in any sequence.

11. Operators with higher precedence are evaluated earlier. For operators of the same precedence, their associativity determines the order of evaluation.

12. All binary operators except assignment operators are left associative; assignment operators are right associative.

QUIZ

Answer the quiz for this chapter online at the Companion Website.

MyLab Programming™

PROGRAMMING EXERCISES

think before coding

Pedagogical Note

For each exercise, carefully analyze the problem requirements and design strategies for solving the problem before coding.

Debugging Tip

Before you ask for help, read and explain the program to yourself, and trace it using several representative inputs by hand or using an IDE debugger. You learn how to program by debugging your own mistakes.

learn from mistakes

Section 3.2

***3.1** (*Algebra: solve quadratic equations*) The two roots of a quadratic equation $ax^2 + bx + c = 0$ can be obtained using the following formula:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

$b^2 - 4ac$ is called the discriminant of the quadratic equation. If it is positive, the equation has two real roots. If it is zero, the equation has one root. If it is negative, the equation has no real roots.

Write a program that prompts the user to enter values for *a*, *b*, and *c* and displays the result based on the discriminant. If the discriminant is positive, display two roots. If the discriminant is `0`, display one root. Otherwise, display "The equation has no real roots."

Note you can use `Math.pow(x, 0.5)` to compute $\sqrt{x}$. Here are some sample runs:

```
Enter a, b, c: 1.0 3 1 ↵Enter
The equation has two roots -0.381966 and -2.61803
```

```
Enter a, b, c: 1 2.0 1 ↵Enter
The equation has one root -1.0
```

```
Enter a, b, c: 1 2 3 ↵Enter
The equation has no real roots
```

3.2 (*Game: add three numbers*) The program in Listing 3.1, AdditionQuiz.java, generates two integers and prompts the user to enter the sum of these two integers. Revise the program to generate three single-digit integers and prompt the user to enter the sum of these three integers.

Sections 3.3–3.7

***3.3** (*Algebra: solve* 2 × 2 *linear equations*) A linear equation can be solved using Cramer's rule given in Programming Exercise 1.13. Write a program that prompts the user to enter *a*, *b*, *c*, *d*, *e*, and *f* and displays the result. If $ad - bc$ is **0**, report that "The equation has no solution."

```
Enter a, b, c, d, e, f: 9.0 4.0 3.0 -5.0 -6.0 -21.0 ↵Enter
x is -2.0 and y is 3.0
```

```
Enter a, b, c, d, e, f: 1.0 2.0 2.0 4.0 4.0 5.0 ↵Enter
The equation has no solution
```

****3.4** (*Random month*) Write a program that randomly generates an integer between 1 and 12 and displays the English month names January, February, . . . , December for the numbers 1, 2, . . . , 12, accordingly.

***3.5** (*Find future dates*) Write a program that prompts the user to enter an integer for today's day of the week (Sunday is 0, Monday is 1, . . . , and Saturday is 6). Also prompt the user to enter the number of days after today for a future day and display the future day of the week. Here is a sample run:

```
Enter today's day: 1 ↵Enter
Enter the number of days elapsed since today: 3 ↵Enter
Today is Monday and the future day is Thursday
```

```
Enter today's day: 0
Enter the number of days elapsed since today: 31 ↵Enter
Today is Sunday and the future day is Wednesday
```

***3.6** (*Health application: BMI*) Revise Listing 3.4, ComputeAndInterpretBMI.java, to let the user enter weight, feet, and inches. For example, if a person is 5 feet and 10 inches, you will enter **5** for feet and **10** for inches. Here is a sample run:

```
Enter weight in pounds: 140 ↵Enter
Enter feet: 5 ↵Enter
Enter inches: 10 ↵Enter
BMI is 20.087702275404553
Normal
```

3.7 (*Financial application: monetary units*) Modify Listing 2.10, ComputeChange.java, to display the nonzero denominations only, using singular words for single units such as 1 dollar and 1 penny, and plural words for more than one unit such as 2 dollars and 3 pennies.

VideoNote
Sort three integers

***3.8** (*Sort three integers*) Write a program that prompts the user to enter three integers and display the integers in non-decreasing order.

****3.9** (*Business: check ISBN-10*) An ISBN-10 (International Standard Book Number) consists of 10 digits: $d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}$. The last digit, d_{10}, is a checksum, which is calculated from the other 9 digits using the following formula:

$$(d_1 \times 1 + d_2 \times 2 + d_3 \times 3 + d_4 \times 4 + d_5 \times 5 +$$
$$d_6 \times 6 + d_7 \times 7 + d_8 \times 8 + d_9 \times 9)\%11$$

If the checksum is **10**, the last digit is denoted as X according to the ISBN-10 convention. Write a program that prompts the user to enter the first 9 digits and displays the 10-digit ISBN (including leading zeros). Your program should read the input as an integer. Here are sample runs:

```
Enter the first 9 digits of an ISBN as integer: 013601267 ↵Enter
The ISBN-10 number is 0136012671
```

```
Enter the first 9 digits of an ISBN as integer: 013031997 ↵Enter
The ISBN-10 number is 013031997X
```

3.10 (*Game: addition quiz*) Listing 3.3, SubtractionQuiz.java, randomly generates a subtraction question. Revise the program to randomly generate an addition question with two integers less than 100.

Sections 3.8–3.16

***3.11** (*Find the number of days in a month*) Write a program that prompts the user to enter the month and year and displays the number of days in the month. For example, if the user entered month **2** and year **2012**, the program should display that February 2012 has 29 days. If the user entered month **3** and year **2015**, the program should display that March 2015 has 31 days.

3.12 (*Palindrome integer*) Write a program that prompts the user to enter a three-digit integer and determines whether it is a palindrome *integer*. An *integer* is palindrome if it reads the same from right to left and from left to right. A negative integer is treated the same as a positive integer. Here are sample runs of this program:

```
Enter a three-digit integer: 121 ↵Enter
121 is a palindrome
```

```
Enter a three-digit integer: 123 ↵Enter
123 is not a palindrome
```

***3.13** (*Financial application: compute taxes*) Listing 3.5, ComputeTax.java, gives the source code to compute taxes for single filers. Complete this program to compute taxes for all filing statuses.

3.14 (*Game: heads or tails*) Write a program that lets the user guess whether the flip of a coin results in heads or tails. The program randomly generates an integer **0** or **1**, which represents head or tail. The program prompts the user to enter a guess, and reports whether the guess is correct or incorrect.

****3.15** (*Game: lottery*) Revise Listing 3.8, Lottery.java, to generate a lottery of a three-digit integer. The program prompts the user to enter a three-digit integer and determines whether the user wins according to the following rules:

1. If the user input matches the lottery number in the exact order, the award is \$10,000.
2. If all digits in the user input match all digits in the lottery number, the award is \$3,000.
3. If one digit in the user input matches a digit in the lottery number, the award is \$1,000.

3.16 (*Random point*) Write a program that displays a random coordinate in a rectangle. The rectangle is centered at (0, 0) with width 100 and height 200.

***3.17** (*Game: scissor, rock, paper*) Write a program that plays the popular scissor–rock–paper game. (A scissor can cut a paper, a rock can knock a scissor, and a paper can wrap a rock.) The program randomly generates a number `0`, `1`, or `2` representing scissor, rock, and paper. The program prompts the user to enter a number `0`, `1`, or `2` and displays a message indicating whether the user or the computer wins, loses, or draws. Here are sample runs:

```
scissor (0), rock (1), paper (2): 1 ↵Enter
The computer is scissor. You are rock. You won
```

```
scissor (0), rock (1), paper (2): 2 ↵Enter
The computer is paper. You are paper too. It is a draw
```

***3.18** (*Cost of shipping*) A shipping company uses the following function to calculate the cost (in dollars) of shipping based on the weight of the package (in pounds).

$$c(w) = \begin{cases} 3.5, \text{ if } 0 < w <= 1 \\ 5.5, \text{ if } 1 < w <= 3 \\ 8.5, \text{ if } 3 < w <= 10 \\ 10.5, \text{ if } 10 < w <= 20 \end{cases}$$

Write a program that prompts the user to enter the weight of the package and displays the shipping cost. If the weight is negative or zero, display a message "Invalid input." If the weight is greater than 20, display a message "The package cannot be shipped."

****3.19** (*Compute the perimeter of a triangle*) Write a program that reads three edges for a triangle and computes the perimeter if the input is valid. Otherwise, display that the input is invalid. The input is valid if the sum of every pair of two edges is greater than the remaining edge.

***3.20** (*Science: wind-chill temperature*) Programming Exercise 2.17 gives a formula to compute the wind-chill temperature. The formula is valid for temperatures in the range between −58°F and 41°F and wind speed greater than or equal to `2`. Write a program that prompts the user to enter a temperature and a wind speed. The program displays the wind-chill temperature if the input is valid; otherwise, it displays a message indicating whether the temperature and/or wind speed is invalid.

Comprehensive

****3.21** (*Science: day of the week*) Zeller's congruence is an algorithm developed by Christian Zeller to calculate the day of the week. The formula is

$$h = \left(q + \frac{26(m + 1)}{10} + k + \frac{k}{4} + \frac{j}{4} + 5j\right)\%7$$

where

- `h` is the day of the week (0: Saturday, 1: Sunday, 2: Monday, 3: Tuesday, 4: Wednesday, 5: Thursday, and 6: Friday).
- `q` is the day of the month.
- `m` is the month (3: March, 4: April, ..., 12: December). January and February are counted as months 13 and 14 of the previous year.
- `j` is $\frac{year}{100}$.
- `k` is the year of the century (i.e., *year* % 100).

Note all divisions in this exercise perform an integer division. Write a program that prompts the user to enter a year, month, and day of the month, and displays the name of the day of the week. Here are some sample runs:

```
Enter year: (e.g., 2012): 2015 ↵Enter
Enter month: 1-12: 1 ↵Enter
Enter the day of the month: 1-31: 25 ↵Enter
Day of the week is Sunday
```

```
Enter year: (e.g., 2012): 2012 ↵Enter
Enter month: 1-12: 5 ↵Enter
Enter the day of the month: 1-31: 12 ↵Enter
Day of the week is Saturday
```

(*Hint*: January and February are counted as 13 and 14 in the formula, so you need to convert the user input 1 to 13 and 2 to 14 for the month and change the year to the previous year. For example, if the user enters 1 for `m` and 2015 for year, `m` will be 13 and `year` will be 2014 used in the formula.)

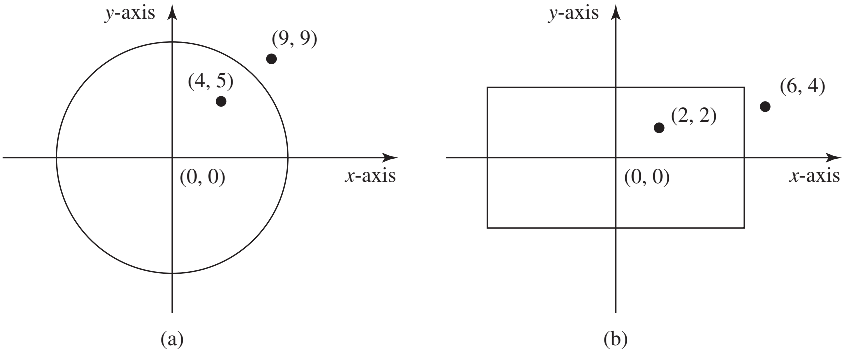
VideoNote
Check point location

****3.22** (*Geometry: point in a circle?*) Write a program that prompts the user to enter a point (`x`, `y`) and checks whether the point is within the circle centered at (`0`, `0`) with radius `10`. For example, (`4`, `5`) is inside the circle and (`9`, `9`) is outside the circle, as shown in Figure 3.7a.

(*Hint*: A point is in the circle if its distance to (`0`, `0`) is less than or equal to `10`. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Test your program to cover all cases.) Two sample runs are shown below:

```
Enter a point with two coordinates: 4 5 ↵Enter
Point (4.0, 5.0) is in the circle
```

```
Enter a point with two coordinates: 9 9 ↵Enter
Point (9.0, 9.0) is not in the circle
```



(a) (b)

FIGURE 3.7 (a) Points inside and outside of the circle. (b) Points inside and outside of the rectangle.

****3.23** (*Geometry: point in a rectangle?*) Write a program that prompts the user to enter a point `(x, y)` and checks whether the point is within the rectangle centered at (`0`, `0`) with width `10` and height `5`. For example, (`2`, `2`) is inside the rectangle and (`6`, `4`) is outside the rectangle, as shown in Figure 3.7b. (*Hint*: A point is in the rectangle if its horizontal distance to (`0`, `0`) is less than or equal to `10 / 2` and its vertical distance to (`0`, `0`) is less than or equal to `5.0 / 2`. Test your program to cover all cases.) Here are two sample runs:

```
Enter a point with two coordinates: -4.9 2.49 ↵Enter
Point (-4.9, 2.49) is in the rectangle
```

```
Enter a point with two coordinates: -5.1 -2.4 ↵Enter
Point (-5.1, -2.4) is not in the rectangle
```

****3.24** (*Game: pick a card*) Write a program that simulates picking a card from a deck of `52` cards. Your program should display the rank (`Ace`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `Jack`, `Queen`, `King`) and suit (`Clubs`, `Diamonds`, `Hearts`, `Spades`) of the card. Here is a sample run of the program:

```
The card you picked is Jack of Hearts
```

***3.25** (*Geometry: intersecting point*) Two points on line 1 are given as (`x1`, `y1`) and (`x2`, `y2`) and on line 2 as (`x3`, `y3`) and (`x4`, `y4`), as shown in Figure 3.8a and b.

The intersecting point of the two lines can be found by solving the following linear equations:

$$\begin{aligned}(y_1 - y_2)x - (x_1 - x_2)y &= (y_1 - y_2)x_1 - (x_1 - x_2)y_1\\(y_3 - y_4)x - (x_3 - x_4)y &= (y_3 - y_4)x_3 - (x_3 - x_4)y_3\end{aligned}$$

This linear equation can be solved using Cramer's rule (see Programming Exercise 3.3). If the equation has no solutions, the two lines are parallel (see

Figure 3.8c). Write a program that prompts the user to enter four points and displays the intersecting point. Here are sample runs:

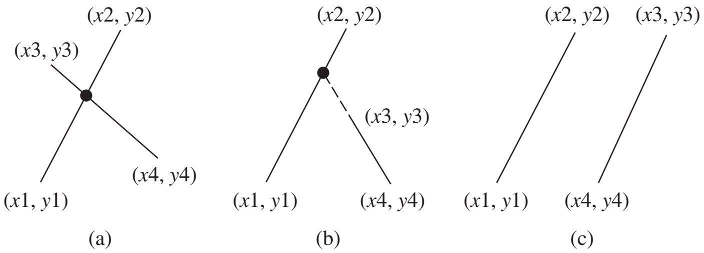


FIGURE 3.8 Two lines intersect in (a and b) and two lines are parallel in (c).

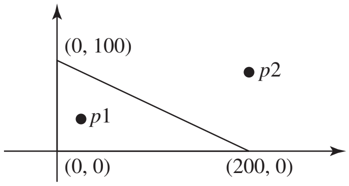
```
Enter x1, y1, x2, y2, x3, y3, x4, y4: 2 2 5 -1.0 4.0 2.0 -1.0 -2.0 ↵Enter
The intersecting point is at (2.88889, 1.1111)
```

```
Enter x1, y1, x2, y2, x3, y3, x4, y4: 2 2 7 6.0 4.0 2.0 -1.0 -2.0 ↵Enter
The two lines are parallel
```

3.26 (*Use the* `&&`, `||`, *and* `^` *operators*) Write a program that prompts the user to enter an integer and determines whether it is divisible by 5 and 6, whether it is divisible by 5 or 6, and whether it is divisible by 5 or 6, but not both. Here is a sample run of this program:

```
Enter an integer: 10 ↵Enter
Is 10 divisible by 5 and 6? false
Is 10 divisible by 5 or 6? true
Is 10 divisible by 5 or 6, but not both? true
```

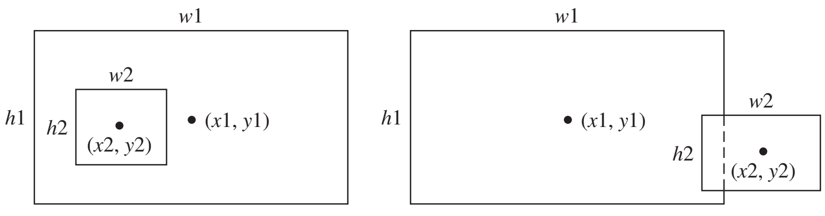
****3.27** (*Geometry: points in triangle?*) Suppose a right triangle is placed in a plane as shown below. The right-angle point is placed at (0, 0), and the other two points are placed at (200, 0) and (0, 100). Write a program that prompts the user to enter a point with *x*- and *y*-coordinates and determines whether the point is inside the triangle. Here are the sample runs:



```
Enter a point's x- and y-coordinates: 100.5 25.5 ↵Enter
The point is in the triangle
```

```
Enter a point's x- and y-coordinates: 100.5 50.5
The point is not in the triangle
```

****3.28** (*Geometry: two rectangles*) Write a program that prompts the user to enter the center *x*-, *y*-coordinates, width, and height of two rectangles and determines whether the second rectangle is inside the first or overlaps with the first, as shown in Figure 3.9. Test your program to cover all cases.



(a) (b)

FIGURE 3.9 (a) A rectangle is inside another one. (b) A rectangle overlaps another one.

Here are the sample runs:

```
Enter r1's center x-, y-coordinates, width, and height: 2.5 4 2.5 43
Enter r2's center x-, y-coordinates, width, and height: 1.5 5 0.5 3
r2 is inside r1
```

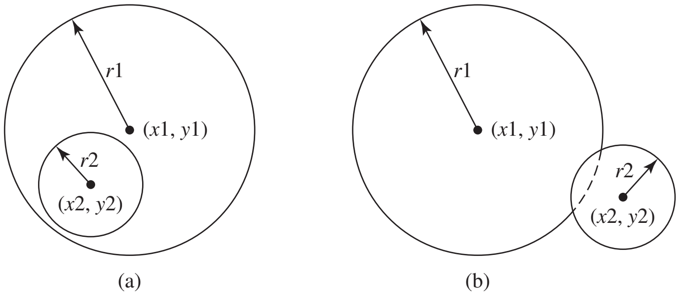
```
Enter r1's center x-, y-coordinates, width, and height: 1 2 3 5.5
Enter r2's center x-, y-coordinates, width, and height: 3 4 4.5 5
r2 overlaps r1
```

```
Enter r1's center x-, y-coordinates, width, and height: 1 2 3 3
Enter r2's center x-, y-coordinates, width, and height: 40 45 3 2
r2 does not overlap r1
```

****3.29** (*Geometry: two circles*) Write a program that prompts the user to enter the center coordinates and radii of two circles and determines whether the second circle is inside the first or overlaps with the first, as shown in Figure 3.10. (*Hint*: `circle2` is inside `circle1` if the distance between the two centers <= `r1 - r2` and `circle2` overlaps `circle1` if the distance between the two centers <= `r1 + r2`. Test your program to cover all cases.)

Here are the sample runs:

```
Enter circle1's center x-, y-coordinates, and radius: 0.5 5.1 13
Enter circle2's center x-, y-coordinates, and radius: 1 1.7 4.5
circle2 is inside circle1
```



(a) (b)

FIGURE 3.10 (a) A circle is inside another circle. (b) A circle overlaps another circle.

```
Enter circle1's center x-, y-coordinates, and radius: 3.4 5.7 5.5 ↵Enter
Enter circle2's center x-, y-coordinates, and radius: 6.7 3.5 3 ↵Enter
circle2 overlaps circle1
```

```
Enter circle1's center x-, y-coordinates, and radius: 3.4 5.5 1 ↵Enter
Enter circle2's center x-, y-coordinates, and radius: 5.5 7.2 1 ↵Enter
circle2 does not overlap circle1
```

***3.30** (*Current time*) Revise Programming Exercise 2.8 to display the hour using a 12-hour clock. Here is a sample run:

```
Enter the time zone offset to GMT: -5 ↵Enter
The current time is 4:50:34 AM
```

***3.31** (*Financials: currency exchange*) Write a program that prompts the user to enter the exchange rate from currency in U.S. dollars to Chinese RMB. Prompt the user to enter `0` to convert from U.S. dollars to Chinese RMB and `1` to convert from Chinese RMB to U.S. dollars. Prompt the user to enter the amount in U.S. dollars or Chinese RMB to convert it to Chinese RMB or U.S. dollars, respectively. Here are the sample runs:

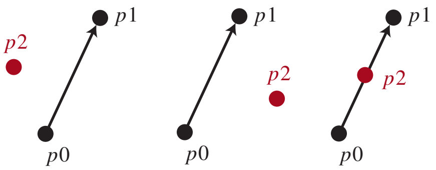
```
Enter the exchange rate from dollars to RMB: 6.81 ↵Enter
Enter 0 to convert dollars to RMB and 1 vice versa: 0 ↵Enter
Enter the dollar amount: 100 ↵Enter
$100.0 is 681.0 yuan
```

```
Enter the exchange rate from dollars to RMB: 6.81 ↵Enter
Enter 0 to convert dollars to RMB and 1 vice versa: 1 ↵Enter
Enter the RMB amount: 10000
10000.0 yuan is $1468.43
```

```
Enter the exchange rate from dollars to RMB: 6.81 ↵Enter
Enter 0 to convert dollars to RMB and 1 vice versa: 5 ↵Enter
CIncorrect input
```

***3.32** (*Geometry: point position*) Given a directed line from point *p*0(*x*0, *y*0) to *p*1(*x*1, *y*1), you can use the following condition to decide whether a point *p*2(*x*2, *y*2) is on the left of the line, on the right, or on the same line (see Figure 3.11):

$$(\mathrm{x1} - \mathrm{x0})*(\mathrm{y2} - \mathrm{y0}) - (\mathrm{x2} - \mathrm{x0})*(\mathrm{y1} - \mathrm{y0}) \begin{cases} >0 & \text{p2 is on the left side of the line} \\ =0 & \text{p2 is on the same line} \\ <0 & \text{p2 is on the right side of the line} \end{cases}$$



(a) (b) (c)

FIGURE 3.11 (a) *p*2 is on the left of the line. (b) *p*2 is on the right of the line. (c) *p*2 is on the same line.

Write a program that prompts the user to enter the three points for *p*0, *p*1, and *p*2 and displays whether *p*2 is on the left of the line from *p*0 to *p*1, to the right, or on the same line. Here are some sample runs:

```
Enter three points for p0, p1, and p2: 4.4 2 6.5 9.5 -5 4 ↵Enter
p2 is on the left side of the line
```

```
Enter three points for p0, p1, and p2: 1 1 5 5 2 2 ↵Enter
p2 is on the same line
```

```
Enter three points for p0, p1, and p2: 3.4 2 6.5 9.5 5 2.5 ↵Enter
p2 is on the right side of the line
```

***3.33** (*Financial: compare costs*) Suppose you shop for rice in two different packages. You would like to write a program to compare the cost. The program prompts the user to enter the weight and price of each package and displays the one with the better price. Here is a sample run:

```
Enter weight and price for package 1: 50 24.59 ↵Enter
Enter weight and price for package 2: 25 11.99 ↵Enter
Package 2 has a better price.
```

```
Enter weight and price for package 1: 50 25 ↵Enter
Enter weight and price for package 2: 25 12.5 ↵Enter
Two packages have the same price.
```

***3.34** (*Geometry: point on line segment*) Exercise 3.32 shows how to test whether a point is on an unbounded line. Revise Exercise 3.32 to test whether a point is on a line segment. Write a program that prompts the user to enter the three points for *p*0, *p*1, and *p*2 and displays whether *p*2 is on the line segment from *p*0 to *p*1. Here are some sample runs:

```
Enter three points for p0, p1, and p2: 1 1 2.5 2.5 1.5 1.5 ↵Enter
(1.5, 1.5) is on the line segment from (1.0, 1.0) to (2.5, 2.5)
```

```
Enter three points for p0, p1, and p2: 1 1 2 2 3.5 3.5 ↵Enter
(3.5, 3.5) is not on the line segment from (1.0, 1.0) to (2.0, 2.0)
```

Note

More than 200 additional programming exercises with solutions are provided to the instructors on the Instructor Resource Website.

CHAPTER

4

MATHEMATICAL FUNCTIONS, CHARACTERS, AND STRINGS

Objectives

- To solve mathematical problems by using the methods in the `Math` class (§4.2).
- To represent characters using the `char` type (§4.3).
- To encode characters using ASCII and Unicode (§4.3.1).
- To represent special characters using the escape sequences (§4.3.2).
- To cast a numeric value to a character and cast a character to an integer (§4.3.3).
- To compare and test characters using the static methods in the `Character` class (§4.3.4).
- To introduce objects and instance methods (§4.4).
- To represent strings using the `String` object (§4.4).
- To return the string length using the `length()` method (§4.4.1).
- To return a character in the string using the `charAt(i)` method (§4.4.2).
- To use the `+` operator to concatenate strings (§4.4.3).
- To return an uppercase string or a lowercase string and to trim a string (§4.4.4).
- To read strings from the console (§4.4.5).
- To read a character from the console (§4.4.6).
- To compare strings using the `equals` and the `compareTo` methods (§4.4.7).
- To obtain substrings (§4.4.8).
- To find a character or a substring in a string using the `indexOf` method (§4.4.9).
- To program using characters and strings (GuessBirthday) (§4.5.1).
- To convert a hexadecimal character to a decimal value (HexDigit2Dec) (§4.5.2).
- To revise the lottery program using strings (LotteryUsingStrings) (§4.5.3).
- To format output using the `System.out.printf` method (§4.6).

4.1 Introduction

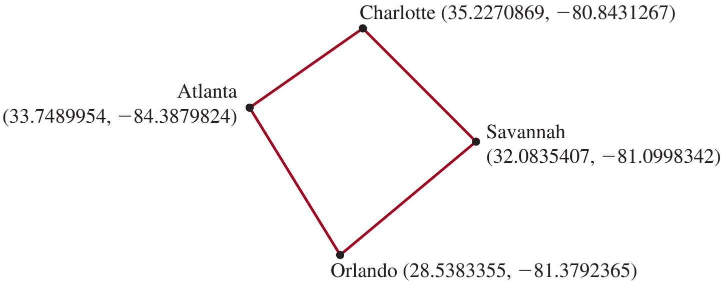
Key Point

The focus of this chapter is to introduce mathematical functions, characters, string objects, and use them to develop programs.

The preceding chapters introduced fundamental programming techniques and taught you how to write simple programs to solve basic problems using selection statements. This chapter introduces methods for performing common mathematical operations. You will learn how to create custom methods in Chapter 6.

problem

Suppose you need to estimate the area enclosed by four cities, given the GPS locations (latitude and longitude) of these cities, as shown in the following diagram. How would you write a program to solve this problem? You will be able to write such a program in this chapter.


Charlotte (35.2270869, −80.8431267)
Atlanta
(33.7489954, −84.3879824)
Savannah
(32.0835407, −81.0998342)
Orlando (28.5383355, −81.3792365)

Because strings are frequently used in programming, it is beneficial to introduce strings early so that you can begin to use them to develop useful programs. This chapter also gives a brief introduction to string objects; you will learn more on objects and strings in Chapters 9 and 10.

4.2 Common Mathematical Functions

Key Point

Java provides many useful methods in the `Math` lass for performing common mathematical functions.

A method is a group of statements that performs a specific task. You have already used the `pow(a, b)` method to compute a^b in Section 2.9.4, Exponent Operations and the random() method for generating a random number in Section 3.7. This section introduces other useful methods in the `Math` class. They can be categorized as *trigonometric methods*, *exponent methods*, and *service methods*. Service methods include the rounding, min, max, absolute, and random methods. In addition to methods, the `Math` class provides two useful `double` constants, `PI` and `E` (the base of natural logarithms). You can use these constants as `Math.PI` and `Math.E` in any program.

4.2.1 Trigonometric Methods

VideoNote
Introduce Math functions

The `Math` class contains the following methods as listed in Table 4.1 for performing trigonometric functions:

The parameter for `sin`, `cos`, and `tan` is an angle in radians. The return value for `asin` and `atan` is an angle in radians in the range between $-\pi/2$ and $\pi/2$, and for `acos` is between 0 and π. One degree is equal to $\pi/180$ in radians, 90 degrees is equal to $\pi/2$ in radians, and 30 degrees is equal to $\pi/6$ in radians.

For example,

```
Math.toDegrees(Math.PI / 2) returns 90.0
Math.toRadians(30) returns 0.5236 (same as π/6)
Math.sin(0) returns 0.0
```

TABLE 4.1 Trigonometric Methods in the Math Class

Method	*Description*
`sin(radians)`	Returns the trigonometric sine of an angle in radians.
`cos(radians)`	Returns the trigonometric cosine of an angle in radians.
`tan(radians)`	Returns the trigonometric tangent of an angle in radians.
`toRadians(degree)`	Returns the angle in radians for the angle in degrees.
`toDegrees(radians)`	Returns the angle in degrees for the angle in radians.
`asin(a)`	Returns the angle in radians for the inverse of sine.
`acos(a)`	Returns the angle in radians for the inverse of cosine.
`atan(a)`	Returns the angle in radians for the inverse of tangent.

`Math.sin(Math.toRadians(270))` returns `-1.0`
`Math.sin(Math.PI / 6)` returns `0.5`
`Math.sin(Math.PI / 2)` returns `1.0`
`Math.cos(0)` returns `1.0`
`Math.cos(Math.PI / 6)` returns `0.866`
`Math.cos(Math.PI / 2)` returns `0`
`Math.asin(0.5)` returns `0.523598333` (same as π/6)
`Math.acos(0.5)` returns `1.0472` (same as π/3)
`Math.atan(1.0)` returns `0.785398` (same as π/4)

4.2.2 Exponent Methods

There are five methods related to exponents in the **`Math`** class as listed in Table 4.2.

TABLE 4.2 Exponent Methods in the Math Class

Method	*Description*
`exp(x)`	Returns e raised to power of x (e^x).
`log(x)`	Returns the natural logarithm of x ($\ln(x) = \log_e(x)$).
`log10(x)`	Returns the base 10 logarithm of x ($\log_{10}(x)$).
`pow(a, b)`	Returns a raised to the power of b (a^b).
`sqrt(x)`	Returns the square root of x ($\sqrt{x}$) for x>=0.

For example,
$e^{3.5}$ is `Math.exp(3.5)`, which returns `33.11545`
`ln(3.5)` is `Math.log(3.5)`, which returns `1.25276`
$\log_{10}$ (3.5) is `Math.log10(3.5)`, which returns `0.544`
2^3 is `Math.pow(2, 3)`, which returns `8.0`
3^2 is `Math.pow(3, 2)`, which returns `9.0`
$4.5^{2.5}$ is `Math.pow(4.5, 2.5)`, which returns `42.9567`
$\sqrt{4}$ is `Math.sqrt(4)`, which returns `2.0`
$\sqrt{10.5}$ is `Math.sqrt(10.5)`, which returns `3.24`

4.2.3 The Rounding Methods

The **`Math`** class contains four rounding methods as listed in Table 4.3.

TABLE 4.3 Rounding Methods in the Math Class

Method	*Description*
`ceil(x)`	*x* is rounded up to its nearest integer. This integer is returned as a double value.
`floor(x)`	*x* is rounded down to its nearest integer. This integer is returned as a double value.
`rint(x)`	*x* is rounded to its nearest integer. If *x* is equally close to two integers, the even one is returned as a double value.
`round(x)`	Returns `(int)Math.floor(x + 0.5)` if *x* is a float and returns `(long)Math.floor(x + 0.5)` if *x* is a double.

For example,

`Math.ceil(2.1)` returns `3.0`
`Math.ceil(2.0)` returns `2.0`
`Math.ceil(-2.0)` returns `-2.0`
`Math.ceil(-2.1)` returns `-2.0`
`Math.floor(2.1)` returns `2.0`
`Math.floor(2.0)` returns `2.0`
`Math.floor(-2.0)` returns `-2.0`
`Math.floor(-2.1)` returns `-3.0`
`Math.rint(2.1)` returns `2.0`
`Math.rint(-2.0)` returns `-2.0`
`Math.rint(-2.1)` returns `-2.0`
`Math.rint(2.5)` returns `2.0`
`Math.rint(3.5)` returns `4.0`
`Math.rint(-2.5)` returns `-2.0`
`Math.round(2.6f)` returns `3 // Returns int`
`Math.round(2.0)` returns `2 // Returns long`
`Math.round(-2.0f)` returns `-2 // Returns int`
`Math.round(-2.6)` returns `-3 // Returns long`
`Math.round(-2.4)` returns `-2 // Returns long`

4.2.4 The min, max, and abs Methods

The **`min`** and **`max`** methods return the minimum and maximum numbers of two numbers (**`int`**, **`long`**, **`float`**, or **`double`**). For example, **`max(4.4, 5.0)`** returns **`5.0`**, and **`min(3, 2)`** returns **`2`**.

The **`abs`** method returns the absolute value of the number (**`int`**, **`long`**, **`float`**, or **`double`**). For example,

`Math.max(2, 3)` returns `3`
`Math.min(2.5, 4.6)` returns `2.5`
`Math.max(Math.max(2.5, 4.6), Math.min(3, 5.6))` returns `4.6`
`Math.abs(-2)` returns `2`
`Math.abs(-2.1)` returns `2.1`

4.2.5 The **random** Method

You used the **`random()`** method in the preceding chapter. This method generates a random **`double`** value greater than or equal to 0.0 and less than 1.0 (**`0 <= Math.random() < 1.0`**). You can use it to write a simple expression to generate random numbers in any range. For example,

`(int)(Math.random() * 10);` → Return a random integer between **`0`** and **`9`**.

`(50 + int)(Math.random() * 50);` → Return a random integer between **`50`** and **`99`**.

In general,

`a + (int)(Math.random() * b);` → Return a random integer between **`a`** and **`a + b - 1`**.

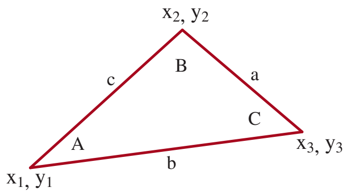
4.2.6 Case Study: Computing Angles of a Triangle

You can use the math methods to solve many computational problems. Given the three sides of a triangle, for example, you can compute the angles by using the following formulas:

$$A = \frac{\text{acos}(a \times a - b \times b - c \times c)}{-2 \times b \times c}$$

$$B = \frac{\text{acos}(b \times b - a \times a - c \times c)}{-2 \times a \times c}$$

$$C = \frac{\text{acos}(c \times c - a \times a - b \times b)}{-2 \times a \times b}$$



Don't be intimidated by the mathematical formula. As we discussed early in Listing 2.9, ComputeLoan.java, you don't have to know how the mathematical formula is derived in order to write a program for computing the loan payments. Here, in this example, given the length of three sides, you can use this formula to write a program to compute the angles without having to know how the formula is derived. In order to compute the lengths of the sides, we need to know the coordinates of three corner points and compute the distances between the points.

Listing 4.1 is an example of a program that prompts the user to enter the *x*- and *y*-coordinates of the three corner points in a triangle then displays the three angles.

LISTING 4.1 `ComputeAngles.java`

```java
import java.util.Scanner;

public class ComputeAngles {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter three points
    System.out.print("Enter three points: ");
    double x1 = input.nextDouble();
    double y1 = input.nextDouble();
    double x2 = input.nextDouble();
    double y2 = input.nextDouble();
    double x3 = input.nextDouble();
    double y3 = input.nextDouble();

    // Compute three sides
    double a = Math.sqrt((x2 - x3) * (x2 - x3)
      + (y2 - y3) * (y2 - y3));
    double b = Math.sqrt((x1 - x3) * (x1 - x3)
      + (y1 - y3) * (y1 - y3));
    double c = Math.sqrt((x1 - x2) * (x1 - x2)
      + (y1 - y2) * (y1 - y2));

    // Compute three angles
    double A = Math.toDegrees(Math.acos((a * a - b * b - c * c)
      / (-2 * b * c)));
    double B = Math.toDegrees(Math.acos((b * b - a * a - c * c)
      / (-2 * a * c)));
    double C = Math.toDegrees(Math.acos((c * c - b * b - a * a)
      / (-2 * a * b)));

    // Display results
    System.out.println("The three angles are " +
      Math.round(A * 100) / 100.0 + " " +
```

enter three points

compute sides

display result

```
35       Math.round(B * 100) / 100.0 + " " +
36       Math.round(C * 100) / 100.0);
37   }
38 }
```

```
Enter three points: 1 1 6.5 1 6.5 2.5 ↵Enter
The three angles are 15.26 90.0 74.74
```

The program prompts the user to enter three points (line 8). This prompting message is not clear. You should give the user explicit instructions on how to enter these points as follows:

```
System.out.print("Enter the coordinates of three points separated "
  + "by spaces like x1 y1 x2 y2 x3 y3: ");
```

Note that the distance between two points `(x1, y1)` and `(x2, y2)` can be computed using the formula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. The program computes the distances between two points (lines 17–22), and applies the formula to compute the angles (lines 25–30). The angles are rounded to display up to two digits after the decimal point (lines 34–36).

The `Math` class is used in the program, but not imported, because it is in the `java.lang` package. All the classes in the `java.lang` package are *implicitly* imported in a Java program.

Check Point

4.2.1 Evaluate the following method calls:

(a) `Math.sqrt(4)`
(b) `Math.sin(2 * Math.PI)`
(c) `Math.cos(2 * Math.PI)`
(d) `Math.pow(2, 2)`
(e) `Math.log(Math.E)`
(f) `Math.exp(1)`
(g) `Math.max(2, Math.min(3, 4))`
(h) `Math.rint(-2.5)`
(i) `Math.ceil(-2.5)`
(j) `Math.floor(-2.5)`
(k) `Math.round(-2.5f)`
(l) `Math.round(-2.5)`
(m) `Math.rint(2.5)`
(n) `Math.ceil(2.5)`
(o) `Math.floor(2.5)`
(p) `Math.round(2.5f)`
(q) `Math.round(2.5)`
(r) `Math.round(Math.abs(-2.5))`

4.2.2 True or false? The argument for trigonometric methods is an angle in radians.

4.2.3 Write a statement that converts `47` degrees to radians and assigns the result to a variable.

4.2.4 Write a statement that converts `PI / 7` to an angle in degrees and assigns the result to a variable.

4.2.5 Write an expression that obtains a random integer between `34` and `55`. Write an expression that obtains a random integer between `0` and `999`. Write an expression that obtains a random number between `5.5` and `55.5`.

4.2.6 Why does the `Math` class not need to be imported?

4.2.7 What is `Math.log(Math.exp(5.5))`?

What is `Math.exp(Math.log(5.5))`?

What is `Math.asin(Math.sin(Math.PI / 6))`?

What is `Math.sin(Math.asin(Math.PI / 6))`?

4.3 Character Data Type and Operations

Key Point

A character data type represents a single character.

char type

In addition to processing numeric values, you can process characters in Java. The character data type, `char`, is used to represent a single character. A character literal is enclosed in single quotation marks. Consider the following code:

```
char letter = 'A';
char numChar = '4';
```

The first statement assigns character `A` to the `char` variable `letter`. The second statement assigns digit character `4` to the `char` variable `numChar`.

Caution

A string literal must be enclosed in double quotation marks (`" "`). A character literal is a single character enclosed in single quotation marks (`' '`). Therefore, `"A"` is a string, but `'A'` is a character.

`char` literal

4.3.1 Unicode and ASCII Code

encoding

Computers use binary numbers internally. A character is stored in a computer as a sequence of 0s and 1s. Mapping a character to its binary representation is called *encoding*. There are different ways to encode a character. How characters are encoded is defined by an *encoding scheme*.

Unicode

original Unicode

Java supports *Unicode*, an encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world's diverse languages. Unicode was originally designed as a 16-bit character encoding. The primitive data type `char` was intended to take advantage of this design by providing a simple data type that could hold any character. However, it turned out that the 65,536 characters possible in a 16-bit encoding are not sufficient to represent all the characters in the world. The Unicode standard therefore has been extended to allow up to 1,112,064 characters. Those characters that go beyond the original 16-bit limit are called *supplementary characters*. Java supports the supplementary characters. The processing and representing of supplementary characters are beyond the scope of this book. For simplicity, this book considers only the original 16-bit Unicode characters. These characters can be stored in a `char` type variable.

supplementary Unicode

A 16-bit Unicode takes two bytes, preceded by `\u`, expressed in four hexadecimal digits that run from `\u0000` to `\uFFFF`. Hexadecimal numbers are introduced in Appendix F, Number Systems. For example, the English word `welcome` is translated into Chinese using two characters, 欢迎. The Unicodes of these two characters are `\u6B22\u8FCE`. The Unicodes for the Greek letters α β γ are `\u03b1 \u03b2 \u03b3` respectively.

ASCII

Most computers use *ASCII* (*American Standard Code for Information Interchange*), an 8-bit encoding scheme, for representing all uppercase and lowercase letters, digits, punctuation marks, and control characters. Unicode includes ASCII code, with `\u0000` to `\u007F` corresponding to the 128 ASCII characters. Table 4.4 shows the ASCII code for some commonly used characters. Appendix B, "The ASCII Character Set," gives a complete list of ASCII characters and their decimal and hexadecimal codes.

TABLE 4.4 ASCII Code for Commonly Used Characters

Characters	*Code Value in Decimal*	*Unicode Value*
`'0'` to `'9'`	48 to 57	`\u0030 to \u0039`
`'A'` to `'Z'`	65 to 90	`\u0041 to \u005A`
`'a'` to `'z'`	97 to 122	`\u0061 to \u007A`

You can use ASCII characters such as `'X'`, `'1'`, and `'$'` in a Java program as well as Unicodes. Thus, for example, the following statements are equivalent:

```
char letter = 'A';
char letter = '\u0041'; // Character A's Unicode is 0041
```

Both statements assign character `A` to the `char` variable `letter`.

Note

`char` increment and decrement

The increment and decrement operators can also be used on `char` variables to get the next or preceding Unicode character. For example, the following statements display character `b`:

```
char ch = 'a';
System.out.println(++ch);
```

4.3.2 Escape Sequences for Special Characters

Suppose you want to print a message with quotation marks in the output. Can you write a statement like this?

```
System.out.println("He said "Java is fun"");
```

No, this statement has a compile error. The compiler thinks the second quotation character is the end of the string and does not know what to do with the rest of the characters.

escape sequence

To overcome this problem, Java uses a special notation to represent special characters, as listed in Table 4.5. This special notation, called an *escape sequence*, consists of a backslash (`\`) followed by a character or a combination of digits. For example, `\t` is an escape sequence for the Tab character, and an escape sequence such as `\u03b1` is used to represent a Unicode. The symbols in an escape sequence are interpreted as a whole rather than individually. An escape sequence is considered as a single character.

Table 4.5 Escape Sequences

Escape Sequence	*Name*	*Unicode Code*	*Decimal Value*
`\b`	Backspace	`\u0008`	8
`\t`	Tab	`\u0009`	9
`\n`	Linefeed	`\u000A`	10
`\f`	Formfeed	`\u000C`	12
`\r`	Carriage Return	`\u000D`	13
`\\`	Backslash	`\u005C`	92
`\"`	Double Quote	`\u0022`	34

So, now you can print the quoted message using the following statement:

```
System.out.println("He said \"Java is fun\"");
```

The output is

```
He said "Java is fun"
```

Note the symbols `\` and `"` together represent one character.

escape character

The backslash `\` is called an *escape character*. It is a special character. To display this character, you have to use an escape sequence `\\`. For example, the following code

```
System.out.println("\\t is a tab character");
```

displays

```
\t is a tab character
```

4.3.3 Casting between `char` and Numeric Types

A `char` can be cast into any numeric type, and vice versa. When an integer is cast into a `char`, only its lower 16 bits of data are used; the other part is ignored. For example:

```
// Note a hex integer is written using prefix 0X
char ch = (char)0XAB0041; // The lower 16 bits hex code 0041 is
                          // assigned to ch
System.out.println(ch);   // ch is character A
```

When a floating-point value is cast into a `char`, the floating-point value is first cast into an `int`, which is then cast into a `char`.

```
char ch = (char)65.25;    // Decimal 65 is assigned to ch
System.out.println(ch);   // ch is character A
```

When a `char` is cast into a numeric type, the character's Unicode is cast into the specified numeric type.

```
int i = (int)'A'; // The Unicode of character A is assigned to i
System.out.println(i);  // i is 65
```

Implicit casting can be used if the result of a casting fits into the target variable. Otherwise, explicit casting must be used. For example, since the Unicode of `'a'` is `97`, which is within the range of a byte, these implicit castings are fine:

```
byte b = 'a';
int i = 'a';
```

But the following statement is incorrect, because the Unicode `\uFFF4` cannot fit into a byte:

```
byte b = '\uFFF4';
```

To force this assignment, use explicit casting, as follows:

```
byte b = (byte)'\uFFF4';
```

Any positive integer between `0` and `FFFF` in hexadecimal can be cast into a character implicitly. Any number not in this range must be cast into a `char` explicitly.

numeric operators on characters

All numeric operators can be applied to `char` operands. A `char` operand is automatically cast into a number if the other operand is a number or a character. If a string is concatenated with a character, the character is converted into a string. For example, the following statements

```
int i = '2' + '3'; // (int)'2' is 50 and (int)'3' is 51
System.out.println("i is " + i); // i is 101
int j = 2 + 'a'; // (int)'a' is 97
System.out.println("j is " + j); // j is 99
System.out.println(j + " is the Unicode for character "
   + (char)j); // 99 is the Unicode for character c
System.out.println("Chapter " + '2');
```

display

```
i is 101
j is 99
99 is the Unicode for character c
Chapter 2
```

4.3.4 Comparing and Testing Characters

Two characters can be compared using the relational operators just like comparing two numbers. This is done by comparing the Unicodes of the two characters. For example,

`'a' < 'b'` is true because the Unicode for `'a'` (`97`) is less than the Unicode for `'b'` (`98`).
`'a' < 'A'` is false because the Unicode for `'a'` (`97`) is greater than the Unicode for `'A'` (`65`).
`'1' < '8'` is true because the Unicode for `'1'` (`49`) is less than the Unicode for `'8'` (`56`).

Often in the program, you need to test whether a character is a number, a letter, an uppercase letter, or a lowercase letter. As given in Appendix B, the ASCII character set, that the Unicodes for lowercase letters are consecutive integers starting from the Unicode for `'a'`, then for `'b'`, `'c'`,. . . , and `'z'`. The same is true for the uppercase letters and for numeric characters. This property can be used to write the code to test characters. For example, the following code tests whether a character `ch` is an uppercase letter, a lowercase letter, or a digital character:

```
if (ch >= 'A' && ch <= 'Z')
  System.out.println(ch + " is an uppercase letter");
else if (ch >= 'a' && ch <= 'z')
  System.out.println(ch + " is a lowercase letter");
else if (ch >= '0' && ch <= '9')
  System.out.println(ch + " is a numeric character");
```

For convenience, Java provides the following methods in the `Character` class for testing characters as listed in Table 4.6. The `Character` class is defined in the `java.lang` package.

TABLE 4.6 Methods in the Character Class

Method	*Description*
`isDigit(ch)`	Returns true if the specified character is a digit.
`isLetter(ch)`	Returns true if the specified character is a letter.
`isLetterOrDigit(ch)`	Returns true if the specified character is a letter or digit.
`isLowerCase(ch)`	Returns true if the specified character is a lowercase letter.
`isUpperCase(ch)`	Returns true if the specified character is an uppercase letter.
`toLowerCase(ch)`	Returns the lowercase of the specified character.
`toUpperCase(ch)`	Returns the uppercase of the specified character.

For example,

```
System.out.println("isDigit('a') is " + Character.isDigit('a'));
System.out.println("isLetter('a') is " + Character.isLetter('a'));
System.out.println("isLowerCase('a') is "
  + Character.isLowerCase('a'));
System.out.println("isUpperCase('a') is "
  + Character.isUpperCase('a'));
System.out.println("toLowerCase('T') is "
  + Character.toLowerCase('T'));
System.out.println("toUpperCase('q') is "
  + Character.toUpperCase('q'));
```

displays

```
isDigit('a') is false
isLetter('a') is true
isLowerCase('a') is true
isUpperCase('a') is false
toLowerCase('T') is t
toUpperCase('q') is Q
```

Check Point

4.3.1 Which of the following are correct literals for characters?

`'1'`, `'\u345dE'`, `'\u3fFa'`, `'\b'`, `'\t'`

4.3.2 How do you display the characters `\` and `"`?

4.3.3 Use print statements to find out the ASCII code for `‘1’`, `‘A’`, `‘B’`, `‘a’`, and `‘b’`. Use print statements to find out the character for the decimal codes `40`, `59`, `79`, `85`, and `90`. Use print statements to find out the character for the hexadecimal code `40`, `5A`, `71`, `72`, and `7A`.

4.3.4 Evaluate the following:

```
int i = '1';
int j = '1' + '2' * ('4' - '3') + 'b' / 'a';
int k = 'a';
char c = 90;
```

4.3.5 Can the following conversions involving casting be allowed? If so, find the converted result.

```
char c = 'A';
int i = (int)c;

float f = 1000.34f;
int i = (int)f;

double d = 1000.34;
int i = (int)d;

int i = 97;
char c = (char)i;
```

4.3.6 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    char x = 'a';
    char y = 'c';
    System.out.println(++x);
    System.out.println(y++);
    System.out.println(x - y);
  }
}
```

4.3.7 Write the code that generates a random lowercase letter.

4.3.8 Show the output of the following statements:

```
System.out.println('a' < 'b');
System.out.println('a' <= 'A');
System.out.println('a' > 'b');
System.out.println('a' >= 'A');
System.out.println('a' == 'a');
System.out.println('a' != 'b');
```

4.4 The String Type

A string is a sequence of characters.

Key Point

The `char` type represents only one character. To represent a string of characters, use the data type called `String`. For example, the following code declares `message` to be a string with the value “`Welcome to Java`".

```
String message = "Welcome to Java";
```

VideoNote

Introduce strings and objects

`String` is a predefined class in the Java library, just like the classes `System` and `Scanner`. The `String` type is not a primitive type. It is known as a *reference type*. Any Java class can be used as a reference type for a variable. The variable declared by a reference type is known as a reference variable that references an object. Here, `message` is a reference variable that references a string object with contents `Welcome to Java`.

Reference data types will be discussed in detail in Chapter 9, Objects and Classes. For the time being, you need to know only how to declare a `String` variable, how to assign a string to the variable, and how to use the methods in the `String` class. More details on using strings will be covered in Chapter 10.

Table 4.7 lists the `String` methods for obtaining string length, for accessing characters in the string, for concatenating string, for converting string to uppercases or lowercases, and for trimming a string.

TABLE 4.7 Simple Methods for `String` Objects

Method	*Description*
`length()`	Returns the number of characters in this string.
`charAt(index)`	Returns the character at the specified index from this string.
`concat(s1)`	Returns a new string that concatenates this string with string `s1`.
`toUpperCase()`	Returns a new string with all letters in uppercase.
`toLowerCase()`	Returns a new string with all letters in lowercase.
`trim()`	Returns a new string with whitespace characters trimmed on both sides.

instance method

static method

Strings are objects in Java. The methods listed in Table 4.7 can only be invoked from a specific string instance. For this reason, these methods are called *instance methods*. A non-instance method is called a *static method*. A static method can be invoked without using an object. All the methods defined in the `Math` class are static methods. They are not tied to a specific object instance. The syntax to invoke an instance method is `referenceVariable.methodName(arguments)`. A method may have many arguments or no arguments. For example, the `charAt(index)` method has one argument, but the `length()` method has no arguments. Recall that the syntax to invoke a static method is `ClassName.methodName(arguments)`. For example, the `pow` method in the `Math` class can be invoked using `Math.pow(2, 2.5)`.

4.4.1 Getting String Length

You can use the `length()` method to return the number of characters in a string. For example, the following code

```
String message = "Welcome to Java";
System.out.println("The length of " + message + " is "
  + message.length());
```

displays

```
The length of Welcome to Java is 15
```

string literal

empty string

Note

When you use a string, you often know its literal value. For convenience, Java allows you to use the *string literal* to refer directly to strings without creating new variables. Thus, `"Welcome to Java".length()` is correct and returns `15`. Note that `""` denotes an *empty string* and `"".length()` is `0`.

4.4.2 Getting Characters from a String

`charAt(index)`

The `s.charAt(index)` method can be used to retrieve a specific character in a string `s`, where the index is between `0` and `s.length()–1`. For example, `message.charAt(0)` returns the character `W`, as shown in Figure 4.1. Note that the index for the first character in the string is `0`.

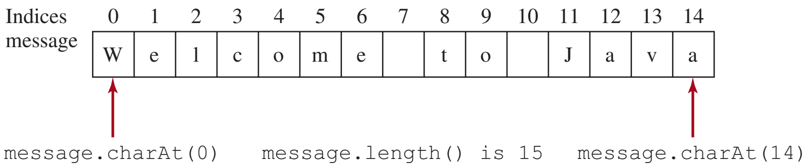


FIGURE 4.1 The characters in a `String` object can be accessed using its index.

Caution

Attempting to access characters in a string `s` out of bounds is a common programming error. To avoid it, make sure that you do not use an index beyond `s.length()–1`. For example, `s.charAt(s.length())` would cause a `StringIndexOutOfBoundsException`.

string index range

4.4.3 Concatenating Strings

You can use the `concat` method to concatenate two strings. The statement given below, for example, concatenates strings `s1` and `s2` into `s3`:

```
String s3 = s1.concat(s2);
```

`s1.concat(s2)`

Because string concatenation is heavily used in programming, Java provides a convenient way to accomplish it. You can use the plus (`+`) operator to concatenate two strings, so the previous statement is equivalent to

```
String s3 = s1 + s2;
```

`s1 + s2`

The following code combines the strings `message`, `" and "`, and `"HTML"` into one string:

```
String myString = message + " and " + "HTML";
```

Recall that the `+` operator can also concatenate a number or a character with a string. In this case, the number or character is converted into a string then concatenated. Note at least one of the operands must be a string in order for concatenation to take place. If one of the operands is a nonstring (e.g., a number), the nonstring value is converted into a string and concatenated with the other string. Here are some examples:

concatenate strings and numbers

```
// Three strings are concatenated
String message = "Welcome " + "to " + "Java";

// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2

// String Supplement is concatenated with character B
String s1 = "Supplement" + 'B'; // s1 becomes SupplementB
```

If neither of the operands is a string, the plus sign (`+`) is the addition operator that adds two numbers.

The augmented `+=` operator can also be used for string concatenation. For example, the following code appends the string `" and Java is fun"` with the string `"Welcome to Java"` in `message`.

```
message += " and Java is fun";
```

So the new `message` is `"Welcome to Java and Java is fun."`

If `i = 1` and `j = 2`, what is the output of the following statement?

```
System.out.println("i + j is " + i + j);
```

The output is `"i + j is 12"` because `"i + j is"` is concatenated with the value of `i` first. To force `i + j` to be executed first, enclose `i + j` in the parentheses, as follows:

```
System.out.println("i + j is " + (i + j));
```

4.4.4 Converting Strings

The `toLowerCase()` method returns a new string with all lowercase letters, and the `toUpperCase()` method returns a new string with all uppercase letters. For example,

`"Welcome".toLowerCase()` returns a new string `welcome`.
`"Welcome".toUpperCase()` returns a new string `WELCOME`.

`toLowerCase()`
`toUpperCase()`

whitespace character

The **`trim()`** method returns a new string by eliminating whitespace characters from both ends of the string. The characters **`' '`**, **`\t`**, **`\f`**, **`\r`**, or **`\n`** are known as *whitespace characters*. For example,

trim()

`"\t Good Night \n".trim()` returns a new string **`Good Night`**.

4.4.5 Reading a String from the Console

read strings

To read a string from the console, invoke the **`next()`** method on a **`Scanner`** object. For example, the following code reads three strings from the keyboard:

```
Scanner input = new Scanner(System.in);
System.out.print("Enter three words separated by spaces: ");
String s1 = input.next();
String s2 = input.next();
String s3 = input.next();
System.out.println("s1 is " + s1);
System.out.println("s2 is " + s2);
System.out.println("s3 is " + s3);
```

```
Enter three words separated by spaces: Welcome to Java ↵Enter
s1 is Welcome
s2 is to
s3 is Java
```

The **`next()`** method reads a string that ends with a whitespace character. You can use the **`nextLine()`** method to read an entire line of text. The **`nextLine()`** method reads a string that ends with the *Enter* key pressed. For example, the following statements read a line of text:

```
Scanner input = new Scanner(System.in);
System.out.println("Enter a line: ");
String s = input.nextLine();
System.out.println("The line entered is " + s);
```

```
Enter a line: Welcome to Java ↵Enter
The line entered is Welcome to Java
```

token-based input

line-based input

For convenience, we call the input using the methods **`next()`**, **`nextByte()`**, **`nextShort()`**, **`nextInt()`**, **`nextLong()`**, **`nextFloat()`**, and **`nextDouble()`** the token-based input, because they read individual elements separated by whitespace characters rather than an entire line. The **`nextLine()`** method is called a line-based input.

avoid input errors

Important Caution

To *avoid input errors*, do not use a line-based input after a token-based input in the program. The reasons will be explained in Section 12.11.4, "How Does **`Scanner`** Work?"

4.4.6 Reading a Character from the Console

To read a character from the console, use the **`nextLine()`** method to read a string and then invoke the **`charAt(0)`** method on the string to return a character. For example, the following code reads a character from the keyboard:

```
Scanner input = new Scanner(System.in);
System.out.print("Enter a character: ");
String s = input.nextLine();
char ch = s.charAt(0);
System.out.println("The character entered is " + ch);
```

4.4.7 Comparing Strings

The `String` class contains the methods, as listed in Table 4.8, for comparing two strings.

TABLE 4.8 Comparison Methods for `String` Objects

Method	*Description*
`equals(s1)`	Returns true if this string is equal to string `s1`.
`equalsIgnoreCase(s1)`	Returns true if this string is equal to string `s1`; it is case insensitive.
`compareTo(s1)`	Returns an integer greater than `0`, equal to `0`, or less than `0` to indicate whether this string is greater than, equal to, or less than `s1`.
`compareToIgnoreCase(s1)`	Same as `compareTo` except that the comparison is case insensitive.
`startsWith(prefix)`	Returns true if this string starts with the specified prefix.
`endsWith(suffix)`	Returns true if this string ends with the specified suffix.
`contains(s1)`	Returns true if `s1` is a substring in this string.

How do you compare the contents of two strings? You might attempt to use the `==` operator, as follows:

`==`

```
if (string1 == string2)
  System.out.println("string1 and string2 are the same object");
else
  System.out.println("string1 and string2 are different objects");
```

However, the `==` operator checks only whether `string1` and `string2` refer to the same object; it does not tell you whether they have the same contents. Therefore, you cannot use the `==` operator to find out whether two string variables have the same contents. Instead, you should use the `equals` method. The following code, for instance, can be used to compare two strings:

`string1.equals(string2)`

```
if (string1.equals(string2))
  System.out.println("string1 and string2 have the same contents");
else
  System.out.println("string1 and string2 are not equal");
```

For example, the following statements display `true` then `false`:

```
String s1 = "Welcome to Java";
String s2 = "Welcome to Java";
String s3 = "Welcome to C++";
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // false
```

The `compareTo` method can also be used to compare two strings. For example, consider the following code:

`s1.compareTo(s2)`

```
s1.compareTo(s2)
```

The method returns the value `0` if `s1` is equal to `s2`, a value less than `0` if `s1` is lexicographically (i.e., in terms of Unicode ordering) less than `s2`, and a value greater than `0` if `s1` is lexicographically greater than `s2`.

The actual value returned from the `compareTo` method depends on the offset of the first two distinct characters in `s1` and `s2` from left to right. For example, suppose `s1` is `abc` and `s2` is `abg`, and `s1.compareTo(s2)` returns `-4`. The first two characters (`a` vs. `a`) from `s1` and `s2` are compared. Because they are equal, the second two characters (`b` vs. `b`) are compared. Because they are also equal, the third two characters (`c` vs. `g`) are compared. Since the character `c` is `4` less than `g`, the comparison returns `-4`.

Caution

Syntax errors will occur if you compare strings by using relational operators `>`, `>=`, `<`, or `<=`. Instead, you have to use `s1.compareTo(s2)`.

Note

The `equals` method returns `true` if two strings are equal, and `false` if they are not. The `compareTo` method returns `0`, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.

The `String` class also provides the `equalsIgnoreCase` and `compareToIgnoreCase` methods for comparing strings. The `equalsIgnoreCase` and `compareToIgnoreCase` methods ignore the case of the letters when comparing two strings. You can also use `str.startsWith(prefix)` to check whether string `str` starts with a specified prefix, `str.endsWith(suffix)` to check whether string `str` ends with a specified suffix, and `str.contains(s1)` to check whether string `str` contains string `s1`. For example,

`"Welcome to Java".startsWith("We")` returns `true`.
`"Welcome to Java".startsWith("we")` returns `false`.
`"Welcome to Java".endsWith("va")` returns `true`.
`"Welcome to Java".endsWith("v")` returns `false`.
`"Welcome to Java".contains("to")` returns `true`.
`"Welcome to Java".contains("To")` returns `false`.

Listing 4.2 gives a program that prompts the user to enter two cities and displays them in alphabetical order.

LISTING 4.2 OrderTwoCities.java

input `city1`

input `city2`

compare two cities

```
 1  import java.util.Scanner;
 2
 3  public class OrderTwoCities {
 4    public static void main(String[] args) {
 5      Scanner input = new Scanner(System.in);
 6
 7      // Prompt the user to enter two cities
 8      System.out.print("Enter the first city: ");
 9      String city1 = input.nextLine();
10      System.out.print("Enter the second city: ");
11      String city2 = input.nextLine();
12
13      if (city1.compareTo(city2) < 0)
14        System.out.println("The cities in alphabetical order are " +
15          city1 + " " + city2);
16      else
17        System.out.println("The cities in alphabetical order are " +
18          city2 + " " + city1);
19    }
20  }
```

```
Enter the first city: New York ↵ Enter
Enter the second city: Boston ↵ Enter
The cities in alphabetical order are Boston New York
```

The program reads two strings for two cities (lines 9 and 11). If `input.nextLine()` is replaced by `input.next()` (line 9), you cannot enter a string with spaces for `city1`. Since a city name may contain multiple words separated by spaces, the program uses the `nextLine` method to read a string (lines 9 and 11). Invoking `city1.compareTo(city2)` compares two strings `city1` with `city2` (line 13). A negative return value indicates that `city1` is less than `city2`.

4.4.8 Obtaining Substrings

You can obtain a single character from a string using the `charAt` method. You can also obtain a substring from a string using the `substring` method (see Figure 4.2) in the `String` class, as given in Table 4.9.

For example,

```
String message = "Welcome to Java";
message = message.substring(0,11) + "HTML";
```

The string `message` now becomes `Welcome` to `HTML`.

TABLE 4.9 The `String` Class Contains the Methods for Obtaining Substrings

Method	*Description*
`substring(beginIndex)`	Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 4.2.
`substring(beginIndex, endIndex)`	Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex – 1`, as shown in Figure 4.2. Note the character at `endIndex` is not part of the substring.

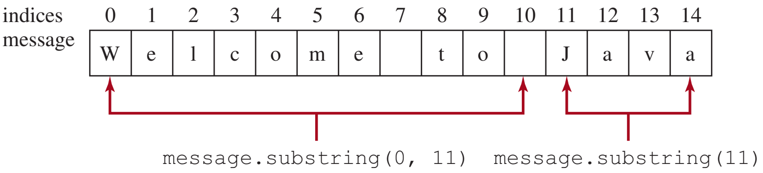


FIGURE 4.2 The `substring` method obtains a substring from a string.

Note

If `beginIndex` is `endIndex`, `substring(beginIndex, endIndex)` returns an empty string with length `0`. If `beginIndex` > `endIndex`, it would be a runtime error.

`beginIndex <= endIndex`

4.4.9 Finding a Character or a Substring in a String

The `String` class provides several versions of `indexOf` and `lastIndexOf` methods to find a character or a substring in a string, as listed in Table 4.10.

TABLE 4.10 The `String` Class Contains the Methods for Finding Substrings

Method	*Description*
`indexOf(ch)`	Returns the index of the first occurrence of `ch` in the string. Returns `-1` if not matched.
`indexOf(ch, fromIndex)`	Returns the index of the first occurrence of `ch` after `fromIndex` in the string. Returns `-1` if not matched.
`indexOf(s)`	Returns the index of the first occurrence of string `s` in this string. Returns `-1` if not matched.
`indexOf(s, fromIndex)`	Returns the index of the first occurrence of string `s` in this string after fromIndex. Returns `-1` if not matched.
`lastIndexOf(ch)`	Returns the index of the last occurrence of `ch` in the string. Returns `-1` if not matched.
`lastIndexOf(ch, fromIndex)`	Returns the index of the last occurrence of `ch` before `fromIndex` in this string. Returns `-1` if not matched.
`lastIndexOf(s)`	Returns the index of the last occurrence of string `s`. Returns `-1` if not matched.
`lastIndexOf(s, fromIndex)`	Returns the index of the last occurrence of string `s` before `fromIndex`. Returns `-1` if not matched.

For example,

`IndexOf`

`"Welcome to Java".indexOf('W')` returns `0`.
`"Welcome to Java".indexOf('o')` returns `4`.
`"Welcome to Java".indexOf('o', 5)` returns `9`.
`"Welcome to Java".indexOf("come")` returns `3`.
`"Welcome to Java".indexOf("Java", 5)` returns `11`.
`"Welcome to Java".indexOf("java", 5)` returns `-1`.

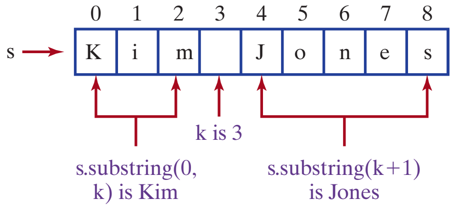
`lastIndexOf`

`"Welcome to Java".lastIndexOf('W')` returns `0`.
`"Welcome to Java".lastIndexOf('o')` returns `9`.
`"Welcome to Java".lastIndexOf('o', 5)` returns `4`.
`"Welcome to Java".lastIndexOf("come")` returns `3`.
`"Welcome to Java".lastIndexOf("Java", 5)` returns `-1`.
`"Welcome to Java".lastIndexOf("Java")` returns `11`.

Suppose that a string `s` contains the first name and last name separated by a space. You can use the following code to extract the first name and last name from the string:

```
int k = s.indexOf(' ');
String firstName = s.substring(0, k);
String lastName = s.substring(k + 1);
```

For example, if `s` is `Kim Jones`, the following diagram illustrates how the first name and last name are extracted.



4.4.10 Conversion between Strings and Numbers

`Integer.parseInt` method

You can convert a numeric string into a number. To convert a string into an `int` value, use the `Integer.parseInt` method, as follows:

```
int intValue = Integer.parseInt(intString);
```

where `intString` is a numeric string such as `"123"`.

`Double.parseDouble` method

To convert a string into a `double` value, use the `Double.parseDouble` method, as follows:

```
double doubleValue = Double.parseDouble(doubleString);
```

where `doubleString` is a numeric string such as `"123.45"`.

If the string is not a numeric string, the conversion would cause a runtime error. The `Integer` and `Double` classes are both included in the `java.lang` package, and thus they are automatically imported.

You can convert a number into a string; simply use the string concatenating operator as follows:

number to string

```
String s = number + "";
```

Check Point

4.4.1 Suppose `s1`, `s2`, and `s3` are three strings, given as follows:

```
String s1 = "Welcome to Java";
String s2 = "Programming is fun";
String s3 = "Welcome to Java";
```

What are the results of the following expressions?

(a) `s1 == s2`
(b) `s2 == s3`
(c) `s1.equals(s2)`
(d) `s1.equals(s3)`
(e) `s1.compareTo(s2)`
(f) `s2.compareTo(s3)`
(g) `s2.compareTo(s2)`
(h) `s1.charAt(0)`
(i) `s1.indexOf('j')`
(j) `s1.indexOf("to")`
(k) `s1.lastIndexOf('a')`
(l) `s1.lastIndexOf("o", 15)`
(m) `s1.length()`
(n) `s1.substring(5)`
(o) `s1.substring(5, 11)`
(p) `s1.startsWith("Wel")`
(q) `s1.endsWith("Java")`
(r) `s1.toLowerCase()`
(s) `s1.toUpperCase()`
(t) `s1.concat(s2)`
(u) `s1.contain(s2)`
(v) `"\t Wel \t".trim()`

4.4.2 Suppose `s1` and `s2` are two strings. Which of the following statements or expressions are incorrect?

```
String s = "Welcome to Java";
String s3 = s1 + s2;
s3 = s1 - s2;
s1 == s2;
s1 >= s2;
s1.compareTo(s2);
int i = s1.length();
char c = s1(0);
char c = s1.charAt(s1.length());
```

4.4.3 Show the output of the following statements (write a program to verify your results):

```
System.out.println("1" + 1);
System.out.println('1' + 1);
System.out.println("1" + 1 + 1);
System.out.println("1" + (1 + 1));
System.out.println('1' + 1 + 1);
```

4.4.4 Evaluate the following expressions (write a program to verify your results):

```
1 + "Welcome " + 1 + 1
1 + "Welcome " + (1 + 1)
1 + "Welcome " + ('\u0001' + 1)
1 + "Welcome " + 'a' + 1
```

4.4.5 Let `s1` be `" Welcome "` and `s2` be `" welcome "`. Write the code for the following statements:

(a) Check whether `s1` is equal to `s2` and assign the result to a Boolean variable `isEqual`.

(b) Check whether `s1` is equal to `s2`, ignoring case, and assign the result to a Boolean variable `isEqual`.

(c) Compare `s1` with `s2` and assign the result to an `int` variable `x`.

(d) Compare `s1` with `s2`, ignoring case, and assign the result to an `int` variable `x`.

(e) Check whether `s1` has the prefix `AAA` and assign the result to a Boolean variable `b`.

(f) Check whether `s1` has the suffix `AAA` and assign the result to a Boolean variable `b`.

(g) Assign the length of `s1` to an `int` variable `x`.
(h) Assign the first character of `s1` to a `char` variable `x`.
(i) Create a new string `s3` that combines `s1` with `s2`.
(j) Create a substring of `s1` starting from index `1`.
(k) Create a substring of `s1` from index `1` to index `4`.
(l) Create a new string `s3` that converts `s1` to lowercase.
(m) Create a new string `s3` that converts `s1` to uppercase.
(n) Create a new string `s3` that trims whitespaces on both ends of `s1`.
(o) Assign the index of the first occurrence of the character `e` in `s1` to an `int` variable `x`.
(p) Assign the index of the last occurrence of the string `abc` in `s1` to an `int` variable `x`.

4.4.6 Write one statement to return the number of digits in an integer `i`.

4.4.7 Write one statement to return the number of digits in a double value `d`.

4.5 Case Studies

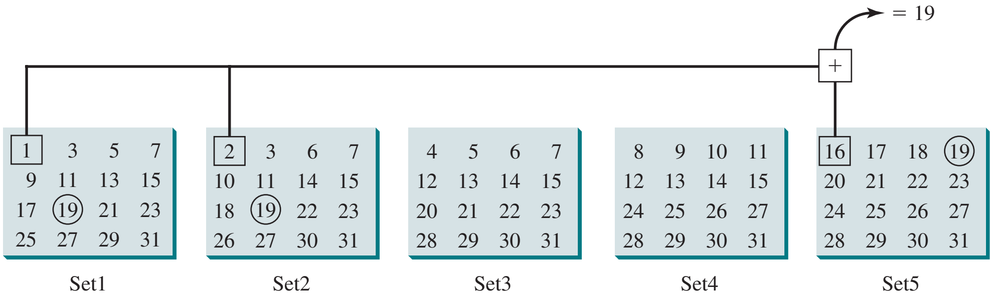
Key Point

Strings are fundamental in programming. The ability to write programs using strings is essential in learning Java programming.

You will frequently use strings to write useful programs. This section presents three examples of solving problems using strings.

4.5.1 Case Study: Guessing Birthdays

You can find out the date of the month when your friend was born by asking five questions. Each question asks whether the day is in one of the five sets of numbers.



1	3	5	7
9	11	13	15
17	19	21	23
25	27	29	31

Set1

2	3	6	7
10	11	14	15
18	19	22	23
26	27	30	31

Set2

4	5	6	7
12	13	14	15
20	21	22	23
28	29	30	31

Set3

8	9	10	11
12	13	14	15
24	25	26	27
28	29	30	31

Set4

16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

Set5

The birthday is the sum of the first numbers in the sets where the day appears. For example, if the birthday is **19**, it appears in Set1, Set2, and Set5. The first numbers in these three sets are **1**, **2**, and **16**. Their sum is **19**.

Listing 4.3 gives a program that prompts the user to answer whether the day is in Set1 (lines 41–44), in Set2 (lines 50–53), in Set3 (lines 59–62), in Set4 (lines 68–71), and in Set5 (lines 77–80). If the number is in the set, the program adds the first number in the set to `day` (lines 47, 56, 65, 74, and 83).

LISTING 4.3 `GuessBirthday.java`

```
 1  import java.util.Scanner;
 2
 3  public class GuessBirthday {
 4    public static void main(String[] args) {
 5      String set1 =
      " 1 3 5 7\n" +
      " 9 11 13 15\n" +
      "17 19 21 23\n" +
      "25 27 29 31";

    String set2 =
      " 2 3 6 7\n" +
      "10 11 14 15\n" +
      "18 19 22 23\n" +
      "26 27 30 31";

    String set3 =
      " 4 5 6 7\n" +
      "12 13 14 15\n" +
      "20 21 22 23\n" +
      "28 29 30 31";

    String set4 =
      " 8 9 10 11\n" +
      "12 13 14 15\n" +
      "24 25 26 27\n" +
      "28 29 30 31";

    String set5 =
      "16 17 18 19\n" +
      "20 21 22 23\n" +
      "24 25 26 27\n" +
      "28 29 30 31";
```

day to be determined

```
    int day = 0;

    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to answer questions
    System.out.print("Is your birthday in Set1?\n");
    System.out.print(set1);
    System.out.print("\nEnter 0 for No and 1 for Yes: ");
    int answer = input.nextInt();
```

in Set1?

```
    if (answer == 1)
      day += 1;

    // Prompt the user to answer questions
    System.out.print("\nIs your birthday in Set2?\n");
    System.out.print(set2);
    System.out.print("\nEnter 0 for No and 1 for Yes: ");
    answer = input.nextInt();
```

in Set2?

```
    if (answer == 1)
      day += 2;

    // Prompt the user to answer questions
    System.out.print("\nIs your birthday in Set3?\n");
    System.out.print(set3);
    System.out.print("\nEnter 0 for No and 1 for Yes: ");
    answer = input.nextInt();
```

in Set3?

```
    if (answer == 1)
      day += 4;
```

```java

    // Prompt the user to answer questions
    System.out.print("\nIs your birthday in Set4?\n");
    System.out.print(set4);
    System.out.print("\nEnter 0 for No and 1 for Yes: ");
    answer = input.nextInt();

```

in Set4?

```java
    if (answer == 1)
      day += 8;

    // Prompt the user to answer questions
    System.out.print("\nIs your birthday in Set5?\n");
    System.out.print(set5);
    System.out.print("\nEnter 0 for No and 1 for Yes: ");
    answer = input.nextInt();

```

in Set5?

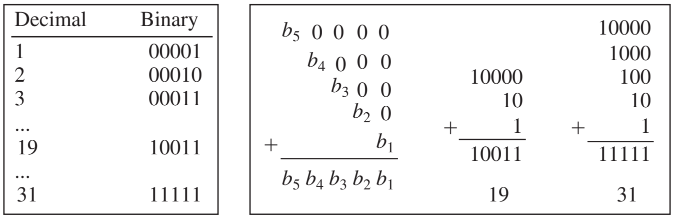
```java
    if (answer == 1)
      day += 16;

    System.out.println("\nYour birthday is " + day + "!");
  }
}
```

```
Is your birthday in Set1?
 1  3  5  7
 9 11 13 15
17 19 21 23
25 27 29 31
Enter 0 for No and 1 for Yes: 1 ↵Enter

Is your birthday in Set2?
 2  3  6  7
10 11 14 15
18 19 22 23
26 27 30 31
Enter 0 for No and 1 for Yes: 1 ↵Enter

Is your birthday in Set3?
 4  5  6  7
12 13 14 15
20 21 22 23
28 29 30 31
Enter 0 for No and 1 for Yes: 0 ↵Enter

Is your birthday in Set4?
 8  9 10 11
12 13 14 15
24 25 26 27
28 29 30 31
Enter 0 for No and 1 for Yes: 0 ↵Enter

Is your birthday in Set5?
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31
Enter 0 for No and 1 for Yes: 1 ↵Enter
Your birthday is 19!
```

line#	day	answer	output
35	0		
44		1	
47	1		
53		1	
56	3		
62		0	
71		0	
80		1	
83	19		
85			Your birthday is 19!

mathematics behind the game

This game is easy to program. You may wonder how the game was created. The mathematics behind the game is actually quite simple. The numbers are not grouped together by accident—the way they are placed in the five sets is deliberate. The starting numbers in the five sets are `1`, `2`, `4`, `8`, and `16`, which correspond to `1`, `10`, `100`, `1000`, and `10000` in binary (binary numbers are introduced in Appendix F, Number Systems). A binary number for decimal integers between `1` and `31` has at most five digits, as shown in Figure 4.3a. Let it be $b_5b_4b_3b_2b_1$. Thus, $b_5b_4b_3b_2b_1 = b_5 0000 + b_4 000 + b_3 00 + b_2 0 + b_1$, as shown in Figure 4.3b. If a day's binary number has a digit `1` in b_k, the number should appear in Set$k$. For example, number `19` is binary `10011`, so it appears in Set1, Set2, and Set5. It is binary `1 + 10 + 10000 = 10011` or decimal `1 + 2 + 16 = 19`. Number `31` is binary `11111`, so it appears in Set1, Set2, Set3, Set4, and Set5. It is binary `1 + 10 + 100 + 1000 + 10000 = 11111` or decimal `1 + 2 + 4 + 8 + 16 = 31`.



Decimal	Binary
1	00001
2	00010
3	00011
...	
19	10011
...	
31	11111

(a)

$$\begin{array}{r} b_5\ 0\ 0\ 0\ 0 \\ b_4\ 0\ 0\ 0 \\ b_3\ 0\ 0 \\ b_2\ 0 \\ +\quad b_1 \\ \hline b_5\ b_4\ b_3\ b_2\ b_1 \end{array} \qquad \begin{array}{r} 10000 \\ 10 \\ +\quad 1 \\ \hline 10011 \\ 19 \end{array} \qquad \begin{array}{r} 10000 \\ 1000 \\ 100 \\ 10 \\ +\quad 1 \\ \hline 11111 \\ 31 \end{array}$$

(b)

Figure 4.3 (a) A number between `1` and `31` can be represented using a five-digit binary number. (b) A five-digit binary number can be obtained by adding binary numbers `1`, `10`, `100`, `1000`, or `10000`.

4.5.2 Case Study: Converting a Hexadecimal Digit to a Decimal Value

The hexadecimal number system has 16 digits: 0–9, A–F. The letters A, B, C, D, E, and F correspond to the decimal numbers 10, 11, 12, 13, 14, and 15. We now write a program that prompts the user to enter a hex digit and display its corresponding decimal value, as given in Listing 4.4.

VideoNote

Convert hex to decimal

Listing 4.4 `HexDigit2Dec.java`

```java
import java.util.Scanner;

public class HexDigit2Dec {
  public static void main(String[] args) {
```

input string

check length

is A–F?

is 0–9?

```
 5       Scanner input = new Scanner(System.in);
 6       System.out.print("Enter a hex digit: ");
 7       String hexString = input.nextLine();
 8
 9       // Check if the hex string has exactly one character
10       if (hexString.length() != 1) {
11         System.out.println("You must enter exactly one character");
12         System.exit(1);
13       }
14
15       // Display decimal value for the hex digit
16       char ch = Character.toUpperCase(hexString.charAt(0));
17       if ('A' <= ch && ch <= 'F') {
18         int value = ch - 'A' + 10;
19         System.out.println("The decimal value for hex digit "
20           + ch + " is " + value);
21       }
22       else if (Character.isDigit(ch)) {
23         System.out.println("The decimal value for hex digit "
24           + ch + " is " + ch);
25       }
26       else {
27         System.out.println(ch + " is an invalid input");
28       }
29     }
30   }
```

```
Enter a hex digit: AB7C ↵Enter
You must enter exactly one character
```

```
Enter a hex digit: B ↵Enter
The decimal value for hex digit B is 11
```

```
Enter a hex digit: 8 ↵Enter
The decimal value for hex digit 8 is 8
```

```
Enter a hex digit: T ↵Enter
T is an invalid input
```

The program reads a string from the console (line 7) and checks if the string contains a single character (line 10). If not, report an error and exit the program (line 12).

The program invokes the `Character.toUpperCase` method to obtain the character `ch` as an uppercase letter (line 16). If `ch` is between `'A'` and `'F'` (line 17), the corresponding decimal value is `ch – 'A' + 10` (line 18). Note `ch – 'A'` is `0` if `ch` is `'A'`, `ch – 'A'` is `1` if `ch` is `'B'`, and so on. When two characters perform a numerical operation, the characters' Unicodes are used in the computation.

The program invokes the `Character.isDigit(ch)` method to check if `ch` is between `'0'` and `'9'` (line 22). If so, the corresponding decimal digit is the same as `ch` (lines 23 and 24).

If `ch` is not between `'A'` and `'F'` nor a digit character, the program displays an error message (line 27).

4.5.3 Case Study: Revising the Lottery Program Using Strings

The lottery program in Listing 3.8, Lottery.java, generates a random two-digit number, prompts the user to enter a two-digit number, and determines whether the user wins according to the following rule:

1. If the user input matches the lottery number in the exact order, the award is $10,000.
2. If all the digits in the user input match all the digits in the lottery number, the award is $3,000.
3. If one digit in the user input matches a digit in the lottery number, the award is $1,000.

The program in Listing 3.8 uses an integer to store the number. Listing 4.5 gives a new program that generates a random two-digit string instead of a number, and receives the user input as a string instead of a number.

LISTING 4.5 `LotteryUsingStrings.java`

```java
import java.util.Scanner;

public class LotteryUsingStrings {
  public static void main(String[] args) {
    // Generate a lottery as a two-digit string
    String lottery = "" + (int)(Math.random() * 10)
      + (int)(Math.random() * 10);

    // Prompt the user to enter a guess
    Scanner input = new Scanner(System.in);
    System.out.print("Enter your lottery pick (two digits): ");
    String guess = input.nextLine();

    // Get digits from lottery
    char lotteryDigit1 = lottery.charAt(0);
    char lotteryDigit2 = lottery.charAt(1);

    // Get digits from guess
    char guessDigit1 = guess.charAt(0);
    char guessDigit2 = guess.charAt(1);

    System.out.println("The lottery number is " + lottery);

    // Check the guess
    if (guess.equals(lottery))
      System.out.println("Exact match: you win $10,000");
    else if (guessDigit2 == lotteryDigit1
          && guessDigit1 == lotteryDigit2)
      System.out.println("Match all digits: you win $3,000");
    else if (guessDigit1 == lotteryDigit1
          || guessDigit1 == lotteryDigit2
          || guessDigit2 == lotteryDigit1
          || guessDigit2 == lotteryDigit2)
      System.out.println("Match one digit: you win $1,000");
    else
      System.out.println("Sorry, no match");
  }
}
```

generate a lottery

enter a guess

exact match?

match one digit?

match all digits?

```
Enter your lottery pick (two digits): 00 ↵Enter
The lottery number is 00
Exact match: you win $10,000
```

```
Enter your lottery pick (two digits): 45 ↵Enter
The lottery number is 54
Match all digits: you win $3,000
```

```
Enter your lottery pick: 23 ↵Enter
The lottery number is 34
Match one digit: you win $1,000
```

```
Enter your lottery pick: 23 ↵Enter
The lottery number is 14
Sorry: no match
```

The program generates two random digits and concatenates them into the string `lottery` (lines 6 and 7). After this, `lottery` contains two random digits.

The program prompts the user to enter a guess as a two-digit string (line 12) and checks the guess against the lottery number in this order:

- First, check whether the guess matches the lottery exactly (line 25).
- If not, check whether the reversal of the guess matches the lottery (line 27).
- If not, check whether one digit is in the lottery (lines 30–33).
- If not, nothing matches and display “Sorry, no match” (line 36).

Check Point

4.5.1 If you run Listing 4.3 GuessBirthday.java with input `1` for Set1, Set3, and Set4 and `0` for Set2 and Set5, what will be the birthday?

4.5.2 If you enter a lowercase letter such as `b`, the program in Listing 4.4 displays `B` is `11`. Revise the code as to display `b` is `11`.

4.5.3 What would be wrong if lines 6 and 7 are in Listing 4.5 replaced by the following code?

```
String lottery = "" + (int)(Math.random() * 100);
```

4.6 Formatting Console Output

Key Point

You can use the `System.out.printf` *method to display formatted output on the console.*

Often, it is desirable to display numbers in a certain format. For example, the following code computes interest, given the amount and the annual interest rate:

```
double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.println("Interest is $" + interest);
```

```
Interest is $16.404674
```

Because the interest amount is currency, it is desirable to display only two digits after the decimal point. To do this, you can write the code as follows:

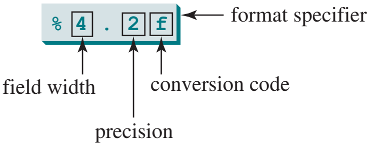
```
double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.println("Interest is $"
  + (int)(interest * 100) / 100.0);
```

```
Interest is $16.4
```

However, the format is still not correct. There should be two digits after the decimal point: `16.40` rather than `16.4`. You can fix it by using the `printf` method, as follows:

printf

```
double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.printf("Interest is $%4.2f",
  interest);
```



```
Interest is $16.40
```

The `f` in the `printf` stands for formatted, implying that the method prints an item in some format. The syntax to invoke this method is

```
System.out.printf(format, item1, item2, ..., itemk);
```

where `format` is a string that may consist of substrings and format specifiers.

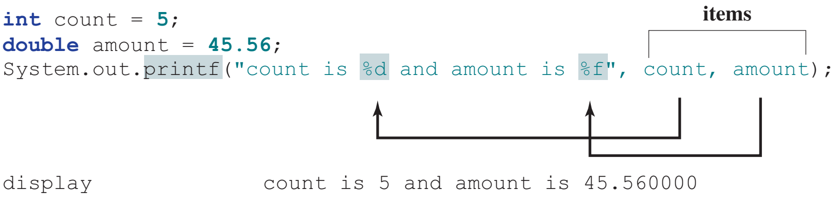
format specifier

A *format specifier* specifies how an item should be formatted. An item may be a numeric value, a character, a Boolean value, or a string. A simple format specifier consists of a percent sign (`%`) followed by a conversion code. Table 4.11 lists some frequently used simple format specifiers.

TABLE 4.11 Frequently Used Format Specifiers

Format Specifier	*Output*	*Example*
`%b`	A Boolean value	True or false
`%c`	A character	‘a’
`%d`	A decimal integer	200
`%f`	A floating-point number	45.460000
`%e`	A number in standard scientific notation	4.556000e+01
`%s`	A string	“Java is cool”

Here is an example:



```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);

display              count is 5 and amount is 45.560000
```

Items must match the format specifiers in order, in number, and in exact type. For example, the format specifier for `count` is `%d` and for `amount` is `%f`. By default, a floating-point value is displayed with six digits after the decimal point. You can specify the width and precision in a format specifier, as shown in the examples in Table 4.12.

TABLE 4.12 Examples of Specifying Width and Precision

Example	*Output*
`%5c`	Output the character and add four spaces before the character item, because the width is 5.
`%6b`	Output the Boolean value and add one space before the false value and two spaces before the true value.
`%5d`	Output the integer item with width 5. If the number of digits in the item is <5, add spaces before the number. If the number of digits in the item is >5, the width is automatically increased.
`%10.2f`	Output the floating-point item with width 10 including a decimal point and two digits after the point. Thus, there are seven digits allocated before the decimal point. If the number of digits before the decimal point in the item is <7, add spaces before the number. If the number of digits before the decimal point in the item is >7, the width is automatically increased.
`%10.2e`	Output the floating-point item with width 10 including a decimal point, two digits after the point and the exponent part. If the displayed number in scientific notation has width <10, add spaces before the number.
`%12s`	Output the string with width 12 characters. If the string item has fewer than 12 characters, add spaces before the string. If the string item has more than 12 characters, the width is automatically increased.

If an item requires more spaces than the specified width, the width is automatically increased. For example, the following code

```
System.out.printf("%3d#%2s#%4.2f\n", 1234, "Java", 51.6653);
```

displays

```
1234#Java#51.67
```

The specified width for `int` item `1234` is `3`, which is smaller than its actual size `4`. The width is automatically increased to `4`. The specified width for string item `Java` is `2`, which is smaller than its actual size `4`. The width is automatically increased to `4`. The specified width for `double` item `51.6653` is `4`, but it needs width 5 to display 51.67, so the width is automatically increased to `5`.

thousand separators

You can display a number with thousand separators by adding a comma in front of a number specifier. For example, the following code

```
System.out.printf("%,8d %,10.1f\n", 12345678, 12345678.263);
```

displays

```
12,345,678 12,345,678.3
```

leading zeros

You can pad a number with leading zeros rather than spaces by adding a `0` in front of a number specifier. For example, the following code

```
System.out.printf("%08d %08.1f\n", 1234, 5.63);
```

displays

```
00001234 000005.6
```

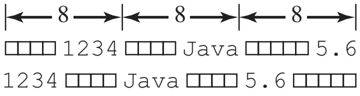
right justify

left justify

By default, the output is right justified. You can put the minus sign (`-`) in the format specifier to specify that the item is left justified in the output within the specified field. For example, the following statements

```
System.out.printf("%8d%8s%8.1f\n", 1234, "Java", 5.63);
System.out.printf("%-8d%-8s%-8.1f \n", 1234, "Java", 5.63);
```

display



where the square box (□) denotes a blank space.

Caution

The items must match the format specifiers in exact type. The item for the format specifier `%f` or `%e` must be a floating-point type value such as `40.0`, not `40`. Thus, an `int` variable cannot match `%f` or `%e`. You can use `%.2f` to specify a floating-point value with two digits after the decimal point. However, `%0.2f` would be incorrect.

Tip

%%

The `%` sign denotes a format specifier. To output a literal `%` in the format string, use `%%`. For example, the following code

```
System.out.printf("%.2f%%\n", 75.234);
```

displays

```
75.23%
```

Listing 4.6 gives a program that uses `printf` to display a table.

LISTING 4.6 FormatDemo.java

```
public class FormatDemo {
  public static void main(String[] args) {
    // Display the header of the table
    System.out.printf("%-10s%-10s%-10s%-10s%-10s\n", "Degrees",
      "Radians", "Sine", "Cosine", "Tangent");

    // Display values for 30 degrees
    int degrees = 30;
    double radians = Math.toRadians(degrees);
    System.out.printf("%-10d%-10.4f%-10.4f%-10.4f%-10.4f\n", degrees,
      radians, Math.sin(radians), Math.cos(radians),
      Math.tan(radians));

    // Display values for 60 degrees
    degrees = 60;
    radians = Math.toRadians(degrees);
    System.out.printf("%-10d%-10.4f%-10.4f%-10.4f%-10.4f\n", degrees,
      radians, Math.sin(radians), Math.cos(radians),
      Math.tan(radians));
  }
}
```

display table header

values for 30 degrees

values for 60 degrees

```
Degrees   Radians   Sine      Cosine    Tangent
30        0.5236    0.5000    0.8660    0.5774
60        1.0472    0.8660    0.5000    1.7321
```

The statements in lines 4 and 5 display the column names of the table. The column names are strings. Each string is displayed using the specifier `%-10s`, which left-justifies the string. The statements in lines 10–12 display the degrees as an integer and four float values. The integer is displayed using the specifier `%-10d`, and each float is displayed using the specifier `%-10.4f`, which specifies four digits after the decimal point.

Check Point

4.6.1 What are the format specifiers for outputting a Boolean value, a character, a decimal integer, a floating-point number, and a string?

4.6.2 What is wrong in the following statements?

(a) `System.out.printf("%5d %d", 1, 2, 3);`

(b) `System.out.printf("%5d %f", 1);`

(c) `System.out.printf("%5d %f", 1, 2);`

(d) `System.out.printf("%.2f\n%0.3f\n", 1.23456, 2.34);`

(e) `System.out.printf("%08s\n", "Java");`

4.6.3 Show the output of the following statements:

(a) `System.out.printf("amount is %f %e\n", 32.32, 32.32);`

(b) `System.out.printf("amount is %5.2f%% %5.4e\n", 32.327, 32.32);`

(c) `System.out.printf("%6b\n", (1 > 2));`

(d) `System.out.printf("%6s\n", "Java");`

(e) `System.out.printf("%-6b%s\n", (1 > 2), "Java");`

(f) `System.out.printf("%6b%-8s\n", (1 > 2), "Java");`

(g) `System.out.printf("%,5d %,6.1f\n", 312342, 315562.932);`

(h) `System.out.printf("%05d %06.1f\n", 32, 32.32);`

KEY TERMS

`char` type, 126
encoding, 127
escape character, 128
escape sequence, 128
format specifier, 147
instance method, 132
line-based input, 134
static method, 132
supplementary Unicode, 127
token-based input, 134
Unicode, 127
whitespace character, 134

CHAPTER SUMMARY

1. Java provides the mathematical methods `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `toRadians`, `toDegrees`, `exp`, `log`, `log10`, `pow`, `sqrt`, `ceil`, `floor`, `rint`, `round`, `min`, `max`, `abs`, and `random` in the `Math` class for performing mathematical functions.
2. The character type `char` represents a single character.
3. An escape sequence consists of a backslash (`\`) followed by a character or a combination of digits.
4. The character `\` is called the escape character.
5. The characters `' '`, `\t`, `\f`, `\r`, and `\n` are known as the whitespace characters.
6. Characters can be compared based on their Unicode using the relational operators.

7. The `Character` class contains the methods `isDigit`, `isLetter`, `isLetterOrDigit`, `isLowerCase`, and `isUpperCase` for testing whether a character is a digit, letter, lowercase, or uppercase. It also contains the `toLowerCase` and `toUpperCase` methods for returning a lowercase or uppercase letter.
8. A *string* is a sequence of characters. A string value is enclosed in matching double quotes (`"`). A character value is enclosed in matching single quotes (`'`).
9. Strings are objects in Java. A method that can only be invoked from a specific object is called an *instance method*. A noninstance method is called a *static method*, which can be invoked without using an object.
10. You can get the length of a string by invoking its `length()` method, retrieve a character at the specified index in the string using the `charAt(index)` method, and use the `indexOf` and `lastIndexOf` methods to find a character or a substring in a string.
11. You can use the `concat` method to concatenate two strings or the plus (`+`) operator to concatenate two or more strings.
12. You can use the `substring` method to obtain a substring from the string.
13. You can use the `equals` and `compareTo` methods to compare strings. The `equals` method returns `true` if two strings are equal, and `false` if they are not equal. The `compareTo` method returns `0`, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.
14. The `printf` method can be used to display a formatted output using format specifiers.

QUIZ

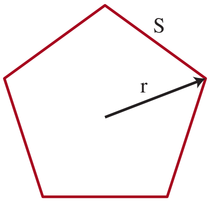
Answer the quiz for this chapter online at the Companion Website.

MyLab Programming™

PROGRAMMING EXERCISES

Section 4.2

4.1 (*Geometry: area of a pentagon*) Write a program that prompts the user to enter the length from the center of a pentagon to a vertex and computes the area of the pentagon, as shown in the following figure.



The formula for computing the area of a pentagon is $Area = \frac{5 \times s^2}{4 \times \tan\left(\frac{\pi}{5}\right)}$, where s is the length of a side. The side can be computed using the formula $s = 2r \sin\frac{\pi}{5}$, where r is the length from the center of a pentagon to a vertex. Round up two digits after the decimal point. Here is a sample run:

```
Enter the length from the center to a vertex: 5.5 ↵Enter
The area of the pentagon is 71.92
```

VideoNote
Compute great circle distance

***4.2** (*Geometry: great circle distance*) The great circle distance is the distance between two points on the surface of a sphere. Let ($x1$, $y1$) and ($x2$, $y2$) be the geographical latitude and longitude of two points. The great circle distance between the two points can be computed using the following formula:

$$d = radius \times \arccos(\sin(x_1) \times \sin(x_2) + \cos(x_1) \times \cos(x_2) \times \cos(y_1 - y_2))$$

Write a program that prompts the user to enter the latitude and longitude of two points on the earth in degrees and displays its great circle distance. The average radius of the earth is 6,371.01 km. Note you need to convert the degrees into radians using the `Math.toRadians` method since the Java trigonometric methods use radians. The latitude and longitude degrees in the formula are for north and west. Use negative to indicate south and east degrees. Here is a sample run:

```
Enter point 1 (latitude and longitude) in degrees: 39.55 -116.25 ↵Enter
Enter point 2 (latitude and longitude) in degrees: 41.5 87.37 ↵Enter
The distance between the two points is 10691.79183231593 km
```

***4.3** (*Geography: estimate areas*) Use the GPS locations for Atlanta, Georgia; Orlando, Florida; Savannah, Georgia; and Charlotte, North Carolina in the figure in Section 4.1 to compute the estimated area enclosed by these four cities. (Hint: Use the formula in Programming Exercise 4.2 to compute the distance between two cities. Divide the polygon into two triangles and use the formula in Programming Exercise 2.19 to compute the area of a triangle.)

4.4 (*Geometry: area of a hexagon*) The area of a hexagon can be computed using the following formula (`s` is the length of a side):

$$Area = \frac{6 \times s^2}{4 \times \tan(\frac{\pi}{6})}$$

Write a program that prompts the user to enter the side of a hexagon and displays its area. Here is a sample run:

```
Enter the side: 5.5 ↵Enter
The area of the hexagon is 78.59
```

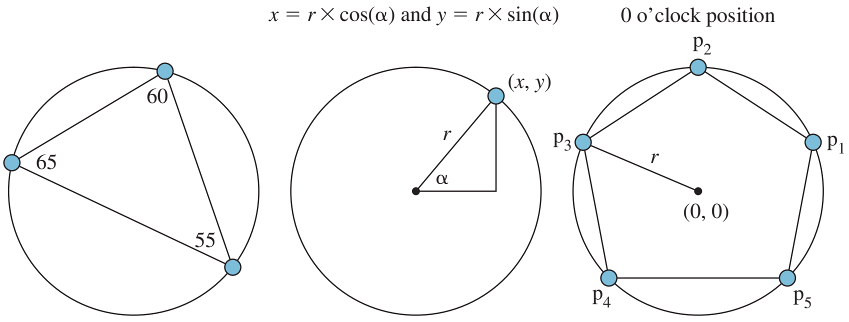
***4.5** (*Geometry: area of a regular polygon*) A regular polygon is an n-sided polygon in which all sides are of the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). The formula for computing the area of a regular polygon is

$$Area = \frac{n \times s^2}{4 \times \tan(\frac{\pi}{n})}$$

Here, s is the length of a side. Write a program that prompts the user to enter the number of sides and their length of a regular polygon and displays its area. Here is a sample run:

```
Enter the number of sides: 5 ↵Enter
Enter the side: 6.5 ↵Enter
The area of the polygon is 72.69017017488385
```

***4.6** (*Random points on a circle*) Write a program that generates three random points on a circle centered at (0, 0) with radius 40 and displays three angles in a triangle formed by these three points, as shown in Figure 4.4a. Display the angles in degrees. (*Hint*: Generate a random angle α in radians between 0 and 2π, as shown in Figure 4.4b and the point determined by this angle is $r^x\cos(\alpha)$, $r^x\sin(\alpha)$.)



(a) (b) (c)

Figure 4.4 (a) A triangle is formed from three random points on the circle. (b) A random point on the circle can be generated using a random angle α. (c) A pentagon is centered at (0, 0) with one point at the 0 o'clock position.

***4.7** (*Corner point coordinates*) Suppose a pentagon is centered at (0, 0) with one point at the 0 o'clock position, as shown in Figure 4.4c. Write a program that prompts the user to enter the radius of the bounding circle of a pentagon and displays the coordinates of the five corner points on the pentagon from p1 to p5 in this order. Use console format to display two digits after the decimal point. Here is a sample run:

```
Enter the radius of the bounding circle: 100.52 ↵Enter
The coordinates of five points on the pentagon are
(95.60, 31.06)
(0.00, 100.52)
(-95.60, 31.06)
(-58.08, -81.32)
(59.08, -81.32)
```

Sections 4.3–4.6

***4.8** (*Find the character of an ASCII code*) Write a program that receives an ASCII code (an integer between `0` and `127`) and displays its character. Here is a sample run:

```
Enter an ASCII code: 69 ↵Enter
The character for ASCII code 69 is E
```

***4.9** (*Find the Unicode of a character*) Write a program that receives a character and displays its Unicode. Here is a sample run:

```
Enter a character: E ↵Enter
The Unicode for the character E is 69
```

***4.10** (*Guess birthday*) Rewrite Listing 4.3, GuessBirthday.java, to prompt the user to enter the character `Y` for Yes and `N` for No, rather than entering `1` for Yes and `0` for No.

***4.11** (*Decimal to hex*) Write a program that prompts the user to enter an integer between `0` and `15` and displays its corresponding hex number. For an incorrect input number, display invalid input. Here are some sample runs:

```
Enter a decimal value (0 to 15): 11 ↵Enter
The hex value is B
```

```
Enter a decimal value (0 to 15): 5 ↵Enter
The hex value is 5
```

```
Enter a decimal value (0 to 15): 31 ↵Enter
31 is an invalid input
```

VideoNote

Convert hex to binary

4.12 (*Hex to binary*) Write a program that prompts the user to enter a hex digit and displays its corresponding binary number in four digits. For example, hex digit `7` is `0111` in binary. Hex digits can be entered either in uppercase or lowercase. For an incorrect input, display invalid input. Here is a sample run:

```
Enter a hex digit: B ↵Enter
The binary value is 1011
```

```
Enter a hex digit: G ↵Enter
G is an invalid input
```

***4.13** (*Vowel or consonant?*) Write a program that prompts the user to enter a letter and check whether the letter is a vowel or consonant. For a nonletter input, display invalid input. Here is a sample run:

```
Enter a letter: B ↵Enter
B is a consonant
```

```
Enter a letter: a ↵Enter
a is a vowel
```

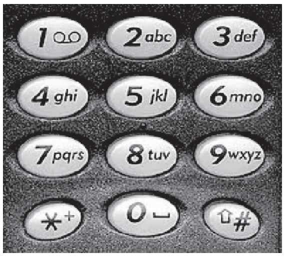
```
Enter a letter: # ↵Enter
# is an invalid input
```

***4.14** (*Convert letter grade to number*) Write a program that prompts the user to enter a letter grade A, B, C, D, or F and displays its corresponding numeric value 4, 3, 2, 1, or 0. For other input, display invalid grade. Here is a sample run:

```
Enter a letter grade: B ↵Enter
The numeric value for grade B is 3
```

```
Enter a letter grade: T ↵Enter
T is an invalid grade
```

***4.15** (*Phone key pads)* The international standard letter/number mapping found on the telephone is shown below:



Write a program that prompts the user to enter a lowercase or uppercase letter and displays its corresponding number. For a nonletter input, display invalid input.

```
Enter a letter: A ↵Enter
The corresponding number is 2
```

```
Enter a letter: a ↵Enter
The corresponding number is 2
```

```
Enter a letter: + ↵Enter
+ is an invalid input
```

4.16 (*Random character*) Write a program that displays a random uppercase letter using the `Math.random()` method.

***4.17** (*Days of a month*) Write a program that prompts the user to enter the year and the first three letters of a month name (with the first letter in uppercase) and displays the number of days in the month. If the input for month is incorrect, display a message as presented in the following sample runs:

```
Enter a year: 2001 ↵Enter
Enter a month: Jan ↵Enter
Jan 2001 has 31 days
```

```
Enter a year: 2016 ↵Enter
Enter a month: jan ↵Enter
jan is not a correct month name
```

***4.18** (*Student major and status*) Write a program that prompts the user to enter two characters and displays the major and status represented in the characters. The first character indicates the major and the second is a number character 1, 2, 3, or 4, which indicates whether a student is a freshman, sophomore, junior, or senior. Suppose that the following characters are used to denote the majors:

M: Mathematics
C: Computer Science
I: Information Technology

Here are sample runs:

```
Enter two characters: M1 ↵Enter
Mathematics Freshman
```

```
Enter two characters: C3 ↵Enter
Computer Science Junior
```

```
Enter two characters: T3 ↵Enter
Invalid input
```

4.19 (*Business: check ISBN-10*) Rewrite Programming Exercise 3.9 by entering the ISBN number as a string.

4.20 (*Process a string*) Write a program that prompts the user to enter a string and displays its length and its first character.

***4.21** (*Check SSN*) Write a program that prompts the user to enter a Social Security number in the format DDD-DD-DDDD, where D is a digit. Your program should check whether the input is valid. Here are sample runs:

```
Enter a SSN: 232-23-5435 ↵Enter
232-23-5435 is a valid social security number
```

```
Enter a SSN: 23-23-5435 ↵Enter
23-23-5435 is an invalid social security number
```

4.22 (*Check substring*) Write a program that prompts the user to enter two strings, and reports whether the second string is a substring of the first string.

```
Enter string s1: ABCD ↵Enter
Enter string s2: BC ↵Enter
BC is a substring of ABCD
```

```
Enter string s1: ABCD ↵Enter
Enter string s2: BDC ↵Enter
BDC is not a substring of ABCD
```

***4.23** (*Financial application: payroll*) Write a program that reads the following information and prints a payroll statement:

Employee's name (e.g., Smith)
Number of hours worked in a week (e.g., 10)
Hourly pay rate (e.g., 9.75)
Federal tax withholding rate (e.g., 20%)
State tax withholding rate (e.g., 9%)

A sample run is as follows:

```
Enter employee's name: Smith ↵Enter
Enter number of hours worked in a week: 10 ↵Enter
Enter hourly pay rate: 9.75 ↵Enter
Enter federal tax withholding rate: 0.20 ↵Enter
Enter state tax withholding rate: 0.09 ↵Enter

Employee Name: Smith
Hours Worked: 10.0
Pay Rate: $9.75
Gross Pay: $97.50
Deductions:
  Federal Withholding (20.0%): $19.50
  State Withholding (9.0%): $8.77
  Total Deduction: $28.27
Net Pay: $69.22
```

***4.24** (*Order three cities*) Write a program that prompts the user to enter three cities and displays them in ascending order. Here is a sample run:

```
Enter the first city: Chicago ↵Enter
Enter the second city: Los Angeles ↵Enter
Enter the third city: Atlanta ↵Enter
The three cities in alphabetical order are Atlanta Chicago
Los Angeles
```

***4.25** (*Generate vehicle plate numbers*) Assume that a vehicle plate number consists of three uppercase letters followed by four digits. Write a program to generate a plate number.

***4.26** (*Financial application: monetary units*) Rewrite Listing 2.10, ComputeChange.java, to fix the possible loss of accuracy when converting a float value to an `int` value. Read the input as a string such as `"11.56"`. Your program should extract the dollar amount before the decimal point, and the cents after the decimal amount using the `indexOf` and `substring` methods.

Note

More than 200 additional programming exercises with solutions are provided to the instructors on the Instructor Resource Website.

CHAPTER
5

LOOPS

Objectives

- To write programs for executing statements repeatedly using a **`while`** loop (§5.2).
- To write loops for the guessing number problem (§5.3).
- To follow the loop design strategy to develop loops (§5.4).
- To control a loop with the user confirmation or a sentinel value (§5.5).
- To obtain large input from a file using input redirection rather than typing from the keyboard (§5.5).
- To write loops using **`do-while`** statements (§5.6).
- To write loops using **`for`** statements (§5.7).
- To discover the similarities and differences of three types of loop statements (§5.8).
- To write nested loops (§5.9).
- To learn the techniques for minimizing numerical errors (§5.10).
- To learn loops from a variety of examples (**`GCD`**, **`FutureTuition`**, and **`Dec2Hex`**) (§5.11).
- To implement program control with **`break`** and **`continue`** (§5.12).
- To process characters in a string using a loop in a case study for checking palindrome (§5.13).
- To write a program that displays prime numbers (§5.14).

5.1 Introduction

A loop can be used to tell a program to execute statements repeatedly.

Key Point

problem

Suppose you need to display a string (e.g., `Welcome to Java!`) a hundred times. It would be tedious to have to write the following statement a hundred times:

100 times

```
System.out.println("Programming is fun");
System.out.println("Programming is fun");
...
System.out.println("Programming is fun");
```

So, how do you solve this problem?

loop

Java provides a powerful construct called a *loop* that controls how many times an operation or a sequence of operations is performed in succession. Using a loop statement, you can simply tell the computer to display a string a hundred times without having to code the print statement a hundred times, as follows:

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count++;
}
```

The variable `count` is initially `0`. The loop checks whether `count < 100` is `true`. If so, it executes the loop body to display the message `Welcome to Java!` and increments `count` by `1`. It repeatedly executes the loop body until `count < 100` becomes `false`. When `count < 100` is `false` (i.e., when `count` reaches `100`), the loop terminates, and the next statement after the loop statement is executed.

Loops are constructs that control repeated executions of a block of statements. The concept of looping is fundamental to programming. Java provides three types of loop statements: `while` loops, `do-while` loops, and `for` loops.

5.2 The while Loop

A `while` loop executes statements repeatedly while the condition is true.

Key Point

The syntax for the `while` loop is as follows:

while loop

```
while (loop-continuation-condition) {
  // Loop body
  Statement(s);
}
```

VideoNote

Use while loop

loop body

iteration

loop-continuation-condition

Figure 5.1a shows the `while` loop flowchart. The part of the loop that contains the statements to be repeated is called the *loop body*. A one-time execution of a loop body is referred to as an *iteration* (or *repetition*) *of the loop*. Each loop contains a `loop-continuation-condition`, a Boolean expression that controls the execution of the body. It is evaluated each time to determine if the loop body is executed. If its evaluation is `true`, the loop body is executed; if its evaluation is `false`, the entire loop terminates and the program control turns to the statement that follows the `while` loop.

The loop for displaying `Welcome to Java!` a hundred times introduced in the preceding section is an example of a `while` loop. Its flowchart is shown in Figure 5.1b.

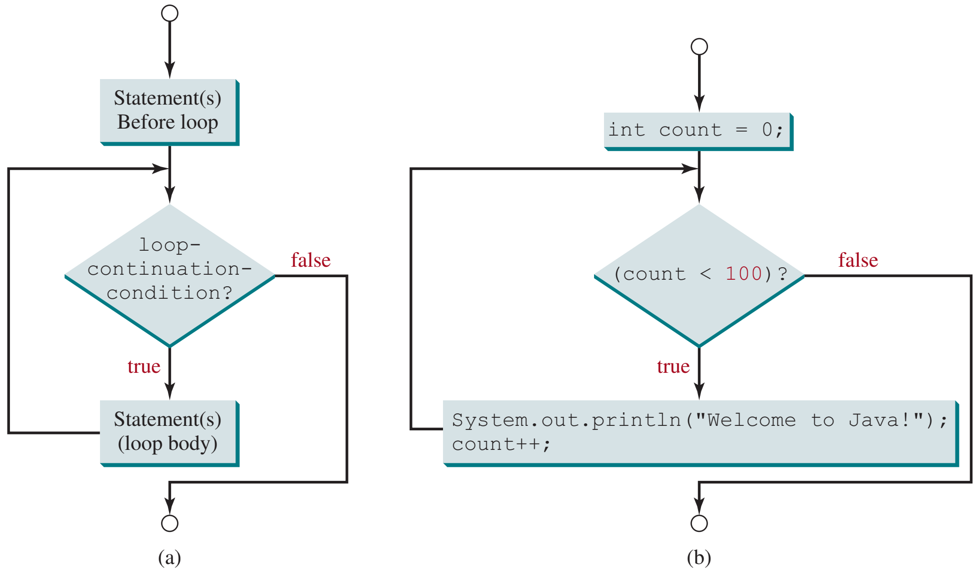
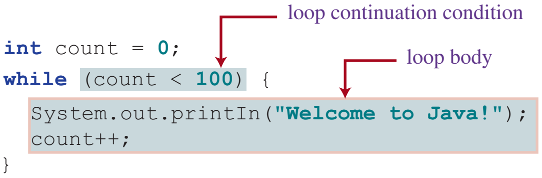

Statement(s)
Before loop
loop-
continuation-
condition?
false
true
Statement(s)
(loop body)
(a)
int count = 0;
(count < 100)?
false
true
System.out.println("Welcome to Java!");
count++;
(b)

FIGURE 5.1 The `while` loop repeatedly executes the statements in the loop body when the `loop-continuation-condition` evaluates to `true`.

The `loop-continuation-condition` is `count < 100` and the loop body contains two statements in the following code:


loop continuation condition
loop body

```
int count = 0;
while (count < 100) {
  System.out.printIn("Welcome to Java!");
  count++;
}
```

In this example, you know exactly how many times the loop body needs to be executed because the control variable `count` is used to count the number of iterations. This type of loop is known as a *counter-controlled loop*.

counter-controlled loop

Note

The `loop-continuation-condition` must always appear inside the parentheses. The braces enclosing the loop body can be omitted only if the loop body contains one or no statement.

Here is another example to help understand how a loop works.

```
int sum = 0, i = 1;
while (i < 10) {
  sum = sum + i;
  i++;
}
System.out.println("sum is " + sum); // sum is 45
```

If `i < 10` is `true`, the program adds `i` to `sum`. Variable `i` is initially set to `1`, then is incremented to `2`, `3`, and up to `10`. When `i` is `10`, `i < 10` is `false`, so the loop exits. Therefore, the sum is `1 + 2 + 3 + ... + 9 = 45`.

What happens if the loop is mistakenly written as follows?

```java
int sum = 0, i = 1;
while (i < 10) {
  sum = sum + i;
}
```

This loop is infinite, because `i` is always `1` and `i < 10` will always be `true`.

infinite loop

Note

Make sure that the `loop-continuation-condition` eventually becomes `false` so that the loop will terminate. A common programming error involves *infinite loops* (i.e., the loop runs forever). If your program takes an unusually long time to run and does not stop, it may have an infinite loop. If you are running the program from the command window, press *CTRL+C* to stop it.

off-by-one error

Caution

Programmers often make the mistake of executing a loop one more or less time. This is commonly known as the *off-by-one error*. For example, the following loop displays `Welcome to Java` 101 times rather than 100 times. The error lies in the condition, which should be `count < 100` rather than `count <= 100`.

```java
int count = 0;
while (count <= 100) {
  System.out.println("Welcome to Java!");
  count++;
}
```

Recall that Listing 3.1, AdditionQuiz.java, gives a program that prompts the user to enter an answer for a question on addition of two single digits. Using a loop, you can now rewrite the program to let the user repeatedly enter a new answer until it is correct, as given in Listing 5.1.

LISTING 5.1 `RepeatAdditionQuiz.java`

generate number1
generate number2
show question
get first answer
check answer
read an answer

```java
import java.util.Scanner;

public class RepeatAdditionQuiz {
  public static void main(String[] args) {
    int number1 = (int)(Math.random() * 10);
    int number2 = (int)(Math.random() * 10);

    // Create a Scanner
    Scanner input = new Scanner(System.in);

    System.out.print(
      "What is " + number1 + " + " + number2 + "? ");
    int answer = input.nextInt();

    while (number1 + number2 != answer)  {
      System.out.print("Wrong answer. Try again. What is "
        + number1 + " + " + number2 + "? ");
      answer = input.nextInt();
    }

    System.out.println("You got it!");
  }
}
```

```
What is 5 + 9? 12 ↵Enter
Wrong answer. Try again. What is 5 + 9? 34 ↵Enter
Wrong answer. Try again. What is 5 + 9? 14 ↵Enter
You got it!
```

The loop in lines 15–19 repeatedly prompts the user to enter an `answer` when `number1 + number2 != answer` is `true`. Once `number1 + number2 != answer` is `false`, the loop exits.

Check Point

5.2.1 Analyze the following code. Is `count < 100` always `true`, always `false`, or sometimes `true` or sometimes `false` at Point A, Point B, and Point C?

```
int count = 0;
while (count < 100) {
  // Point A
  System.out.println("Welcome to Java!");
  count++;
  // Point B
}
// Point C
```

5.2.2 How many times are the following loop bodies repeated? What is the output of each loop?

```
int i = 1;
while (i < 10)
  if (i % 2 == 0)
    System.out.println(i);
```

(a)

```
int i = 1;
while (i < 10)
  if (i % 2 == 0)
    System.out.println(i++);
```

(b)

```
int i = 1;
while (i < 10)
  if ((i++) % 2 == 0)
    System.out.println(i);
```

(c)

5.2.3 What is the output of the following code? Explain the reason.

```
int x = 80000000;

while (x > 0)
  x++;

System.out.println("x is " + x);
```

5.3 Case Study: Guessing Numbers

Key Point

This case study generates a random number and lets the user repeatedly guess a number until it is correct.

VideoNote

Guess a number

The problem is to guess what number a computer has in mind. You will write a program that randomly generates an integer between `0` and `100`, inclusive. The program prompts the user to enter a number continuously until the number matches the randomly generated number. For each user input, the program tells the user whether the input is too low or too high, so the user can make the next guess intelligently. Here is a sample run:

```
Guess a magic number between 0 and 100
Enter your guess: 50 ↵Enter
Your guess is too high
Enter your guess: 25 ↵Enter
Your guess is too low
Enter your guess: 42 ↵Enter
Your guess is too high
Enter your guess: 39 ↵Enter
Yes, the number is 39
```

intelligent guess

The magic number is between `0` and `100`. To minimize the number of guesses, enter `50` first. If your guess is too high, the magic number is between `0` and `49`. If your guess is too low, the magic number is between `51` and `100`. Thus, you can eliminate half of the numbers from further consideration after one guess.

think before coding

How do you write this program? Do you immediately begin coding? No. It is important to *think before coding*. Think how you would solve the problem without writing a program. You need first to generate a random number between `0` and `100`, inclusive, then to prompt the user to enter a guess, then to compare the guess with the random number.

code incrementally

It is a good practice to *code incrementally* one step at a time. For programs involving loops, if you don't know how to write a loop right away, you may first write the code for executing the loop one time, then figure out how to repeatedly execute the code in a loop. For this program, you may create an initial draft, as given in Listing 5.2.

LISTING 5.2 GuessNumberOneTime.java

generate a number

enter a guess

correct guess

too high

too low

```
 1  import java.util.Scanner;
 2
 3  public class GuessNumberOneTime {
 4    public static void main(String[] args) {
 5      // Generate a random number to be guessed
 6      int number = (int)(Math.random() * 101);
 7
 8      Scanner input = new Scanner(System.in);
 9      System.out.println("Guess a magic number between 0 and 100");
10
11      // Prompt the user to guess the number
12      System.out.print("\nEnter your guess: ");
13      int guess = input.nextInt();
14
15      if (guess == number)
16        System.out.println("Yes, the number is " + number);
17      else if (guess > number)
18        System.out.println("Your guess is too high");
19      else
20        System.out.println("Your guess is too low");
21    }
22  }
```

When you run this program, it prompts the user to enter a guess only once. To let the user enter a guess repeatedly, you may wrap the code in lines 11–20 in a loop as follows:

```
while (true) {
  // Prompt the user to guess the number
  System.out.print("\nEnter your guess: ");
  guess = input.nextInt();

  if (guess == number)
    System.out.println("Yes, the number is " + number);
  else if (guess > number)
    System.out.println("Your guess is too high");
  else
    System.out.println("Your guess is too low");
} // End of loop
```

This loop repeatedly prompts the user to enter a guess. However, this loop is not correct, because it never terminates. When `guess` matches `number`, the loop should end. Thus, the loop can be revised as follows:

```
while (guess != number) {
  // Prompt the user to guess the number
```

```
    System.out.print("\nEnter your guess: ");
    guess = input.nextInt();

    if (guess == number)
      System.out.println("Yes, the number is " + number);
    else if (guess > number)
      System.out.println("Your guess is too high");
    else
      System.out.println("Your guess is too low");
  } // End of loop
```

The complete code is given in Listing 5.3.

Listing 5.3 GuessNumber.java

```
 1  import java.util.Scanner;
 2
 3  public class GuessNumber {
 4    public static void main(String[] args) {
 5      // Generate a random number to be guessed
 6      int number = (int)(Math.random() * 101);
 7
 8      Scanner input = new Scanner(System.in);
 9      System.out.println("Guess a magic number between 0 and 100");
10
11      int guess = -1;
12      while (guess != number) {
13        // Prompt the user to guess the number
14        System.out.print("\nEnter your guess: ");
15        guess = input.nextInt();
16
17        if (guess == number)
18          System.out.println("Yes, the number is " + number);
19        else if (guess > number)
20          System.out.println("Your guess is too high");
21        else
22          System.out.println("Your guess is too low");
23      } // End of loop
24    }
25  }
```

generate a number

enter a guess

too high

too low

	line#	number	guess	output
	6	39		
	11		-1	
iteration 1	15		50	
	20			Your guess is too high
iteration 2	15		25	
	22			Your guess is too low
iteration 3	15		42	
	20			Your guess is too high
iteration 4	15		39	
	18			Yes, the number is 39

The program generates the magic number in line 6 and prompts the user to enter a guess continuously in a loop (lines 12–23). For each guess, the program checks whether the guess is correct, too high, or too low (lines 17–22). When the guess is correct, the program exits the loop (line 12). Note that `guess` is initialized to `-1`. Initializing it to a value between `0` and `100` would be wrong, because that could be the number to be guessed.

Check Point

5.3.1 What is wrong if `guess` is initialized to `0` in line 11 in Listing 5.3?

5.4 Loop Design Strategies

Key Point

The key to designing a loop is to identify the code that needs to be repeated and write a condition for terminating the loop.

Writing a correct loop is not an easy task for novice programmers. Consider three steps when writing a loop.

Step 1: Identify the statements that need to be repeated.

Step 2: Wrap these statements in a loop as follows:

```
while (true) {
  Statements;
}
```

Step 3: Code the `loop-continuation-condition` and add appropriate statements for controlling the loop.

```
while (loop-continuation-condition) {
  Statements;
  Additional statements for controlling the loop;
}
```

VideoNote

Multiple subtraction quiz

The Math subtraction learning tool program in Listing 3.3, SubtractionQuiz.java, generates just one question for each run. You can use a loop to generate questions repeatedly. How do you write the code to generate five questions? Follow the loop design strategy. First, identify the statements that need to be repeated. These are the statements for obtaining two random numbers, prompting the user with a subtraction question, and grading the question. Second, wrap the statements in a loop. Third, add a loop control variable and the `loop-continuation-condition` to execute the loop five times.

Listing 5.4 gives a program that generates five questions and, after a student answers all five, reports the number of correct answers. The program also displays the time spent on the test and lists all the questions.

LISTING 5.4 SubtractionQuizLoop.java

get start time

loop

```
import java.util.Scanner;

public class SubtractionQuizLoop {
  public static void main(String[] args) {
    final int NUMBER_OF_QUESTIONS = 5; // Number of questions
    int correctCount = 0; // Count the number of correct answers
    int count = 0; // Count the number of questions
    long startTime = System.currentTimeMillis();
    String output = " "; // output string is initially empty
    Scanner input = new Scanner(System.in);

    while (count < NUMBER_OF_QUESTIONS) {
      // 1. Generate two random single-digit integers
      int number1 = (int)(Math.random() * 10);
      int number2 = (int)(Math.random() * 10);

      // 2. If number1 < number2, swap number1 with number2
      if (number1 < number2) {
        int temp = number1;
        number1 = number2;
        number2 = temp;
      }

      // 3. Prompt the student to answer "What is number1 – number2?"
      System.out.print(
        "What is " + number1 + " - " + number2 + "? ");
      int answer = input.nextInt();

      // 4. Grade the answer and display the result
      if (number1 - number2 == answer) {
        System.out.println("You are correct!");
        correctCount++; // Increase the correct answer count
      }
      else
        System.out.println("Your answer is wrong.\n" + number1
          + " - " + number2 + " should be " + (number1 - number2));

      // Increase the question count
      count++;

      output += "\n" + number1 + "-" + number2 + "=" + answer +
        ((number1 - number2 == answer) ? " correct": " wrong");
    }

    long endTime = System.currentTimeMillis();
    long testTime = endTime - startTime;

    System.out.println("Correct count is " + correctCount +
      "\nTest time is " + testTime / 1000 + " seconds\n" + output);
  }
}
```

display a question

grade an answer

increase correct count

increase control variable

prepare output

end loop

get end time

test time

display result

```
What is 9 - 2? 7
You are correct!

What is 3 - 0? 3
You are correct!

What is 3 - 2? 1
You are correct!

What is 7 - 4? 4
Your answer is wrong.
7 - 4 should be 3

What is 7 - 5? 4
Your answer is wrong.
7 - 5 should be 2

Correct count is 3
Test time is 1021 seconds

9-2=7 correct
3-0=3 correct
3-2=1 correct
7-4=4 wrong
7-5=4 wrong
```

The program uses the control variable **`count`** to control the execution of the loop. **`count`** is initially **`0`** (line 7) and is increased by **`1`** in each iteration (line 39). A subtraction question is displayed and processed in each iteration. The program obtains the time before the test starts in line 8 and the time after the test ends in line 45, then computes the test time in line 46. The test time is in milliseconds and is converted to seconds in line 49.

Check Point

5.4.1 Revise the code using the **`System.nanoTime()`** to measure the time in nano seconds.

5.5 Controlling a Loop with User Confirmation or a Sentinel Value

Key Point

It is a common practice to use a sentinel value to terminate the input.

The preceding example executes the loop five times. If you want the user to decide whether to continue, you can offer a user *confirmation*. The template of the program can be coded as follows:

```
char continueLoop = 'Y';
while (continueLoop == 'Y') {
  // Execute the loop body once
  ...
  // Prompt the user for confirmation
  System.out.print("Enter Y to continue and N to quit: ");
  continueLoop = input.getLine().charAt(0);
}
```

You can rewrite the program given in Listing 5.4 with user confirmation to let the user decide whether to advance to the next question.

sentinel value

sentinel-controlled loop

Another common technique for controlling a loop is to designate a special value when reading and processing a set of values. This special input value, known as a *sentinel value*, signifies the end of the input. A loop that uses a sentinel value to control its execution is called a *sentinel-controlled loop.*

Listing 5.5 gives a program that reads and calculates the sum of an unspecified number of integers. The input **`0`** signifies the end of the input. Do you need to declare a new variable for each input value? No. Just use one variable named **`data`** (line 12) to store the input value, and use a variable named **`sum`** (line 15) to store the total. Whenever a value is read, assign it to **`data`** and, if it is not zero, add it to **`sum`** (line 17).

LISTING 5.5 SentinelValue.java

input

loop

```
 1  import java.util.Scanner;
 2
 3  public class SentinelValue {
 4    /** Main method */
 5    public static void main(String[] args) {
 6      // Create a Scanner
 7      Scanner input = new Scanner(System.in);
 8
 9      // Read an initial data
10      System.out.print(
11        "Enter an integer (the input ends if it is 0): ");
12      int data = input.nextInt();
13
14      // Keep reading data until the input is 0
15      int sum = 0;
16      while (data != 0) {
17        sum += data;
18
19        // Read the next data
20        System.out.print(
21          "Enter an integer (the input ends if it is 0): ");
22        data = input.nextInt();
23      }
24
25      System.out.println("The sum is " + sum);
26    }
27  }
```

end of loop

display result

```
Enter an integer (the input ends if it is 0): 2 ↵Enter
Enter an integer (the input ends if it is 0): 3 ↵Enter
Enter an integer (the input ends if it is 0): 4 ↵Enter
Enter an integer (the input ends if it is 0): 0 ↵Enter
The sum is 9
```

	line#	data	sum	output
	12	2		
	15		0	
iteration 1	17		2	
	22	3		
iteration 2	17		5	
	22	4		
iteration 3	17		9	
	22	0		
	25			The sum is 9

If `data` is not `0`, it is added to `sum` (line 17) and the next item of input data is read (lines 20–22). If `data` is `0`, the loop body is no longer executed and the `while` loop terminates. The input value `0` is the sentinel value for this loop. Note if the first input read is `0`, the loop body never executes, and the resulting sum is `0`.

Caution

Don't use floating-point values for equality checking in a loop control. Because floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results.

Consider the following code for computing `1 + 0.9 + 0.8 + ... + 0.1`:

```
double item = 1; double sum = 0;
while (item != 0) { // No guarantee item will be 0
  sum += item;
  item -= 0.1;
}
System.out.println(sum);
```

Variable `item` starts with `1` and is reduced by `0.1` every time the loop body is executed. The loop should terminate when `item` becomes `0`. However, there is no guarantee that item will be exactly `0`, because the floating-point arithmetic is approximated. This loop seems okay on the surface, but it is actually an infinite loop.

numeric error

In the preceding example, if you have a large number of data to enter, it would be cumbersome to type from the keyboard. You can store the data separated by whitespaces in a text file, say **input.txt**, and run the program using the following command:

```
java SentinelValue < input.txt
```

input redirection

This command is called *input redirection*. The program takes the input from the file **input.txt** rather than having the user type the data from the keyboard at runtime. Suppose the contents of the file are as follows:

```
2 3 4 5 6 7 8 9 12 23 32
23 45 67 89 92 12 34 35 3 1 2 4 0
```

The program should get `sum` to be `518`.

output redirection

Similarly, there is *output redirection*, which sends the output to a file rather than displaying it on the console. The command for output redirection is

```
java ClassName > output.txt
```

Input and output redirections can be used in the same command. For example, the following command gets input from **input.txt** and sends output to **output.txt**:

```
java SentinelValue < input.txt > output.txt
```

Try running the program to see what contents are in **output.txt**.

When reading data through input redirection, you can invoke `input.hasNext()` to detect the end of input. For example, the following code reads all `int` value from the input and displays their total.

```
import java.util.Scanner;

public class TestEndOfInput {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    int sum = 0;

    while (input.hasNext ()) {
      sum += input.nextInt();
    }

    System.out.println(“The sum is “ + sum);
  }
}
```

If there is no more input in the file, `input.hasNext()` will return `false`.

Note

If you enter the input from the command window, you can end the input by pressing ENTER and then CTRL+Z, and then pressing ENTER again. In this case, `input.hasNext()` will return `false`.

Check Point

5.5.1 Suppose the input is `2 3 4 5 0`. What is the output of the following code?

```
import java.util.Scanner;

public class Test {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int number, max;
    number = input.nextInt(); max = number;

    while (number != 0) {
      number = input.nextInt();
      if (number > max)
        max = number;
    }

    System.out.println("max is " + max);
    System.out.println("number " + number);
  }
}
```

5.6 The `do-while` Loop

A `do-while` loop is the same as a `while` loop except that it executes the loop body first then checks the loop continuation condition.

Key Point

VideoNote

Use do-while loop

do-while loop

The `do-while` *loop* is a variation of the `while` loop. Its syntax is as follows:

```
do {
  // Loop body;
  Statement(s);
} while (loop-continuation-condition);
```

Its execution flowchart is shown in Figure 5.2a.

The loop body is executed first, then the `loop-continuation-condition` is evaluated. If the evaluation is `true`, the loop body is executed again; if it is `false`, the `do-while` loop terminates. For example, the following `while` loop statement

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count++;
}
```

can be written using a `do-while` loop as follows:

```
int count = 0;
do {
  System.out.println("Welcome to Java!");
  count++;
} while (count < 100);
```

The flowchart of this `do-while` loop is shown in Figure 5.2b.

The difference between a `while` loop and a `do-while` loop is the order in which the `loop-continuation-condition` is evaluated and the loop body is executed. In the case of a `do-while` loop, the loop body is executed at least once. You can write a loop using either the `while` loop or the `do-while` loop. Sometimes one is a more convenient choice than the other. For example, you can rewrite the `while` loop in Listing 5.5 using a `do-while` loop, as given in Listing 5.6.

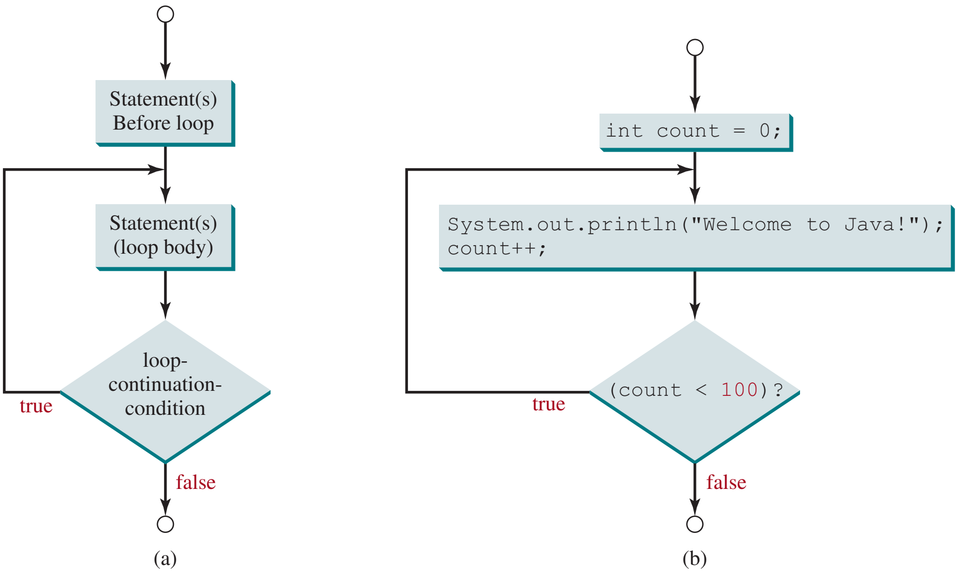


FIGURE 5.2 The `do-while` loop executes the loop body first then checks the `loop-continuation-condition` to determine whether to continue or terminate the loop.

LISTING 5.6 TestDoWhile.java

loop

end loop

```java
import java.util.Scanner;

public class TestDoWhile {
  /** Main method */
  public static void main(String[] args) {
    int data;
    int sum = 0;

    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Keep reading data until the input is 0
    do {
      // Read the next data
      System.out.print(
        "Enter an integer (the input ends if it is 0): ");
      data = input.nextInt();

      sum += data;
    } while (data != 0);

    System.out.println("The sum is " + sum);
  }
}
```

```
Enter an integer (the input ends if it is 0): 3 ↵Enter
Enter an integer (the input ends if it is 0): 5 ↵Enter
Enter an integer (the input ends if it is 0): 6 ↵Enter
Enter an integer (the input ends if it is 0): 0 ↵Enter
The sum is 14
```

Tip

Use a `do-while` loop if you have statements inside the loop that must be executed *at least once*, as in the case of the `do-while` loop in the preceding `TestDoWhile` program. These statements must appear before the loop as well as inside it if you use a `while` loop.

Check Point

5.6.1 Suppose the input is `2 3 4 5 0`. What is the output of the following code?

```java
import java.util.Scanner;

public class Test {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    int number, max;
    number = input.nextInt();
    max = number;

    do {
      number = input.nextInt();
      if (number > max)
        max = number;
    } while (number != 0);
```

```
        System.out.println("max is " + max);
        System.out.println("number " + number);
      }
    }
```

5.6.2 What are the differences between a `while` loop and a `do-while` loop? Convert the following `while` loop into a `do-while` loop:

```
Scanner input = new Scanner(System.in);
int sum = 0;
System.out.println("Enter an integer " +
   "(the input ends if it is 0)");
int number = input.nextInt();
while (number != 0) {
  sum += number;
  System.out.println("Enter an integer " +
     "(the input ends if it is 0)");
  number = input.nextInt();
}
```

5.7 The `for` Loop

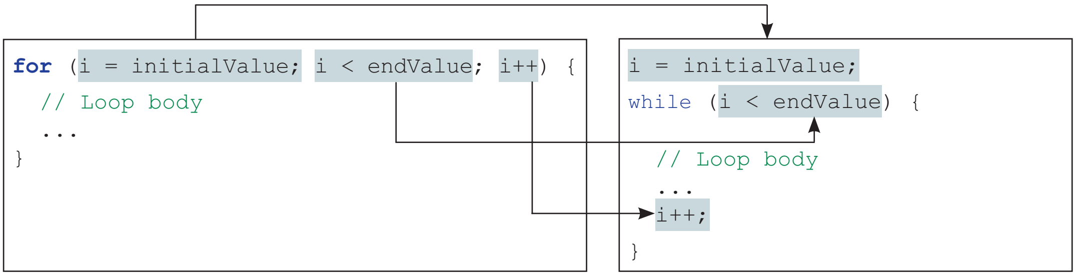
A `for` loop has a concise syntax for writing loops.

Key Point

Often you write a loop in the following common form:

```
i = initialValue; // Initialize loop control variable
while (i < endValue) {
  // Loop body
  ...
  i++; // Adjust loop control variable
}
```

This loop is intuitive and easy for beginners to grasp. However, programmers often forget to adjust the control variable, which leads to an infinite loop. A `for` loop can be used to avoid the potential error and simplify the preceding loop as shown in (a) below. In general, the syntax for a for loop is as shown in (a), which is equivalent to (b).



```
for (i = initialValue; i < endValue; i++) {
  // Loop body
  ...
}
```

(a)

```
i = initialValue;
while (i < endValue) {
  // Loop body
  ...
  i++;
}
```

(b)

In general, the syntax of a `for` *loop* is as follows:

for loop

```
for (initial-action; loop-continuation-condition;
     action-after-each-iteration) {
  // Loop body;
  Statement(s);
}
```

The flowchart of the `for` loop is shown in Figure 5.3a.

The `for` loop statement starts with the keyword `for`, followed by a pair of parentheses enclosing the control structure of the loop. This structure consists of `initial-action`, `loop-continuation-condition`, and `action-after-each-iteration`. The control structure is

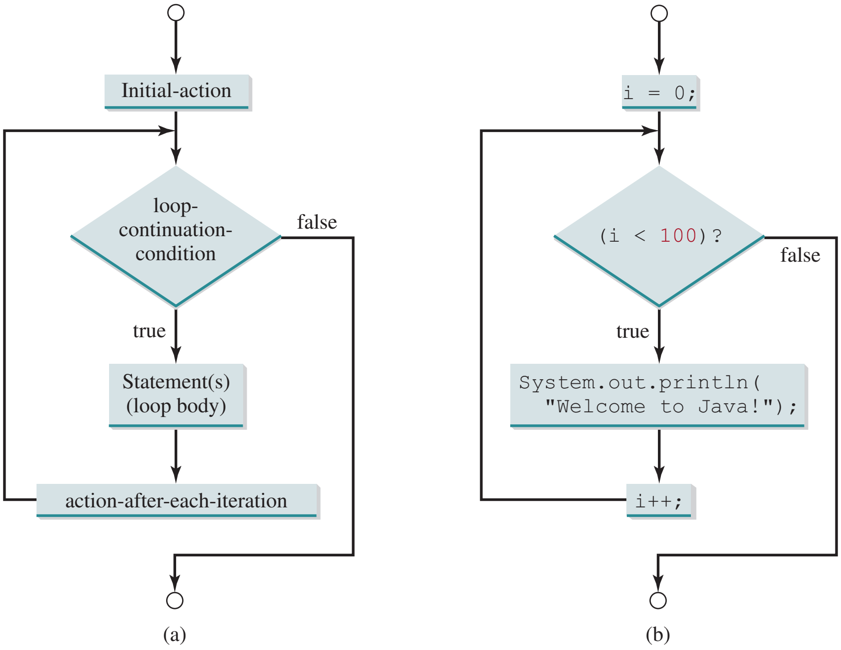

Initial-action
loop-continuation-condition
false
true
Statement(s) (loop body)
action-after-each-iteration
(a)
i = 0;
(i < 100)?
false
true
System.out.println("Welcome to Java!");
i++;
(b)

FIGURE 5.3 A `for` loop performs an initial action once, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the `loop-continuation-condition` evaluates to `true`.

followed by the loop body enclosed inside braces. The `initial-action`, `loop-continuation-condition`, and `action-after-each-iteration` are separated by semicolons.

control variable

A `for` loop generally uses a variable to control how many times the loop body is executed and when the loop terminates. This variable is referred to as a *control variable*. The `initial-action` often initializes a control variable, the `action-after-each-iteration` usually increments or decrements the control variable, and the `loop-continuation-condition` tests whether the control variable has reached a termination value. For example, the following `for` loop prints `Welcome to Java!` a hundred times:

```
int i;
for (i = 0; i < 100; i++) {
  System.out.println("Welcome to Java!");
}
```

The flowchart of the statement is shown in Figure 5.3b. The `for` loop initializes `i` to `0`, then repeatedly executes the `println` statement and evaluates `i++` while `i` is less than `100`.

initial-action

The `initial-action`, `i = 0`, initializes the control variable, `i`. The `loop-continuation-condition`, `i < 100`, is a Boolean expression. The expression is evaluated right after the initialization and at the beginning of each iteration. If this condition is `true`, the loop body is executed. If it is `false`, the loop terminates and the program control turns to the line following the loop.

action-after-each-iteration

The `action-after-each-iteration`, `i++`, is a statement that adjusts the control variable. This statement is executed after each iteration and increments the control variable. Eventually, the value of the control variable should force the `loop-continuation-condition` to become `false`; otherwise, the loop is infinite.

The loop control variable can be declared and initialized in the `for` loop. Here is an example:

```
for (int i = 0; i < 100; i++) {
  System.out.println("Welcome to Java!");
}
```

omitting braces

If there is only one statement in the loop body, as in this example, the braces can be omitted.

declare control variable

Tip

The control variable must be declared inside the control structure of the loop or before the loop. If the loop control variable is used only in the loop, and not elsewhere, it is a good programming practice to declare it in the `initial-action` of the `for` loop. If the variable is declared inside the loop control structure, it cannot be referenced outside the loop. In the preceding code, for example, you cannot reference `i` outside the `for` loop, because it is declared inside the `for` loop.

for loop variations

Note

The `initial-action` in a `for` loop can be a list of zero or more comma-separated variable declaration statements or assignment expressions. For example:

```
for (int i = 0, j = 0; i + j < 10; i++, j++) {
  // Do something
}
```

The `action-after-each-iteration` in a `for` loop can be a list of zero or more comma-separated statements. For example:

```
for (int i = 1; i < 100; System.out.println(i), i++) ;
```

This example is correct, but it is a bad example, because it makes the code difficult to read. Normally, you declare and initialize a control variable as an initial action, and increment or decrement the control variable as an action after each iteration.

Note

If the `loop-continuation-condition` in a `for` loop is omitted, it is implicitly `true`. Thus, the statement given below in (a), which is an infinite loop, is the same as in (b). To avoid confusion, though, it is better to use the equivalent loop in (c).

```
for ( ; ; ) {
  // Do something
}
```

(a)

Equivalent

```
for ( ; true; ) {
  // Do something
}
```

(b)

Equivalent

This is better

```
while (true) {
  // Do something
}
```

(c)

Check Point

5.7.1 Do the following two loops result in the same value in `sum`?

```
for (int i = 0; i < 10; ++i) {
  sum += i;
}
```

(a)

```
for (int i = 0; i < 10; i++) {
  sum += i;
}
```

(b)

5.7.2 What are the three parts of a `for` loop control? Write a `for` loop that prints the numbers from `1` to `100`.

5.7.3 Suppose the input is `2 3 4 5 0`. What is the output of the following code?

```
import java.util.Scanner;

public class Test {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    int number, sum = 0, count;
    for (count = 0; count < 5; count++) {
      number = input.nextInt();
      sum += number;
    }
```

```
      System.out.println("sum is " + sum);
      System.out.println("count is " + count);
    }
  }
```

5.7.4 What does the following statement do?

```
for ( ; ; ) {
  // Do something
}
```

5.7.5 If a variable is declared in a `for` loop control, can it be used after the loop exits?

5.7.6 Convert the following `for` loop statement to a `while` loop and to a `do-while` loop:

```
long sum = 0;
for (int i = 0; i <= 1000; i++)
  sum = sum + i;
```

5.7.7 Count the number of iterations in the following loops.

```
int count = 0;
while (count < n) {
  count++;
}
```

(a)

```
for (int count = 0;
  count <= n; count++) {
}
```

(b)

```
int count = 5;
while (count < n) {
  count++;
}
```

(c)

```
int count = 5;
while (count < n) {
  count = count + 3;
}
```

(d)

5.8 Which Loop to Use?

Key Point

You can use a `for` *loop, a* `while` *loop, or a* `do-while` *loop, whichever is convenient.*

The `while` loop and `do-while` loop are easier to learn than the `for` loop. However, you will learn the `for` loop quickly after some practice. A `for` loop places control variable initialization, loop continuation condition, and adjustment after each iteration all together. It is more concise and enables you to write the code with less errors than the other two loops.

pretest loop

posttest loop

The `while` loop and `for` loop are called *pretest loops* because the continuation condition is checked before the loop body is executed. The `do-while` loop is called a *posttest loop* because the condition is checked after the loop body is executed. The three forms of loop statements—`while`, `do-while`, and `for`—are expressively equivalent; that is, you can write a loop in any of these three forms. For example, a `while` loop in (a) in the following figure can always be converted into the `for` loop in (b).

```
while (loop-continuation-condition) {
  // Loop body
}
```

(a)

Equivalent

```
for ( ; loop-continuation-condition; ) {
  // Loop body
}
```

(b)

A `for` loop in (a) in the next figure can generally be converted into the `while` loop in (b) except in certain special cases (see CheckPoint Question 5.12.2 in Section 5.12 for such a case).

```
for (initial-action;
  loop-continuation-condition;
  action-after-each-iteration) {
  // Loop body;
}
```

(a)

Equivalent

```
initial-action;
while (loop-continuation-condition) {
  // Loop body;
  action-after-each-iteration;
}
```

(b)

Use the loop statement that is most intuitive and comfortable for you. In general, a `for` loop may be used if the number of repetitions is known in advance, as, for example, when you need to display a message a hundred times. A `while` loop may be used if the number of repetitions is not fixed, as in the case of reading the numbers until the input is `0`. A `do-while` loop can be used to replace a `while` loop if the loop body has to be executed before the continuation condition is tested.

Caution

Adding a semicolon at the end of the `for` clause before the loop body is a common mistake, as shown below in (a). In (a), the semicolon signifies the end of the loop prematurely. The loop body is actually empty, as shown in (b). (a) and (b) are equivalent. Both are incorrect.

Error

```
for (int i = 0; i < 10; i++);
{
  System.out.println("i is " + i);
}
```

(a)

Empty body

```
for (int i = 0; i < 10; i++) { };
{
  System.out.println("i is " + i);
}
```

(b)

Similarly, the loop in (c) is also wrong. (c) is equivalent to (d). Both are incorrect.

Error

```
int i = 0;
while (i < 10);
{
  System.out.println("i is " + i);
  i++;
}
```

(c)

Empty body

```
int i = 0;
while (i < 10) { };
{
  System.out.println("i is " + i);
  i++;
}
```

(d)

These errors often occur when you use the next-line block style. Using the end-of-line block style can avoid errors of this type.

In the case of the `do-while` loop, the semicolon is needed to end the loop.

```
int i = 0;
do {
  System.out.println("i is " + i);
  i++;
} while (i < 10);
```

This is correct.

Check Point

5.8.1 Can you convert a `for` loop to a `while` loop? List the advantages of using `for` loops.

5.8.2 Can you always convert a `while` loop into a `for` loop? Convert the following `while` loop into a `for` loop:

```
int i = 1;
int sum = 0;
while (sum <  10000) {
  sum = sum + i;
  i++;
}
```

5.8.3 Identify and fix the errors in the following code:

```java
public class Test {
  public void main(String[] args) {
    for (int i = 0; i < 10; i++);
      sum += i;

    if (i < j);
      System.out.println(i)
    else
      System.out.println(j);

    while (j < 10);
    {
      j++;
    }

    do {
      j++;
    } while (j < 10)
  }
}
```

5.8.4 What is wrong with the following programs?

```java
public class ShowErrors {
  public static void main(String[] args) {
    int i = 0;
    do  {
      System.out.println(i + 4);
      i++;
    }
    while (i < 10)
  }
}
```

(a)

```java
public class ShowErrors {
  public static void main(String[] args) {
    for (int i = 0; i < 10; i++);
      System.out.println(i + 4);
  }
}
```

(b)

5.9 Nested Loops

Key Point

A loop can be nested inside another loop.

Nested loops consist of an outer loop and one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered, and started anew.

nested loop

Listing 5.7 presents a program that uses nested `for` loops to display a multiplication table.

Listing 5.7 MultiplicationTable.java

table title

```java
public class MultiplicationTable {
  /** Main method */
  public static void main(String[] args) {
    // Display the table heading
    System.out.println("         Multiplication Table");

    // Display the number title
    System.out.print("   ");
    for (int j = 1; j <= 9; j++)
      System.out.print("   " + j);
11
12       System.out.println("\n-----------------------------------------");
13
14       // Display table body
15       for (int i = 1; i <= 9; i++) {
16         System.out.print(i + " | ");
17         for (int j = 1; j <= 9; j++) {
18           // Display the product and align properly
19           System.out.printf("%4d", i * j);
20         }
21         System.out.println();
22       }
23     }
24   }
```

outer loop

inner loop

```
Multiplication Table
        1   2   3   4   5   6   7   8   9
-----------------------------------------
1 |    1   2   3   4   5   6   7   8   9
2 |    2   4   6   8  10  12  14  16  18
3 |    3   6   9  12  15  18  21  24  27
4 |    4   8  12  16  20  24  28  32  36
5 |    5  10  15  20  25  30  35  40  45
6 |    6  12  18  24  30  36  42  48  54
7 |    7  14  21  28  35  42  49  56  63
8 |    8  16  24  32  40  48  56  64  72
9 |    9  18  27  36  45  54  63  72  81
```

The program displays a title (line 5) on the first line in the output. The first **for** loop (lines 9 and 10) displays the numbers **1–9** on the second line. A dashed (**–**) line is displayed on the third line (line 12).

The next loop (lines 15–22) is a nested **for** loop with the control variable **i** in the outer loop and **j** in the inner loop. For each **i**, the product `i * j` is displayed on a line in the inner loop, with **j** being **1**, **2**, **3**, …, **9**.

Note

Be aware that a nested loop may take a long time to run. Consider the following loop nested in three levels:

```
for (int i = 0; i < 10000; i++)
  for (int j = 0; j < 10000; j++)
    for (int k = 0; k < 10000; k++)
      Perform an action
```

The action is performed one trillion times. If it takes 1 microsecond to perform the action, the total time to run the loop would be more than 277 hours. Note 1 microsecond is one-millionth (10^{-6}) of a second.

Check Point

5.9.1 How many times is the `println` statement executed?

```
for (int i = 0; i < 10; i++)
  for (int j = 0; j < i; j++)
    System.out.println(i * j)
```

5.9.2 Show the output of the following programs. (*Hint*: Draw a table and list the variables in the columns to trace these programs.)

```
public class Test {
  public static void main(String[] args) {
    for (int i = 1; i < 5; i++) {
      int j = 0;
      while (j < i) {
        System.out.print(j + " ");
        j++;
      }
    }
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    int i = 0;
    while (i < 5) {
      for (int j = i; j > 1; j--)
        System.out.print(j + " ");
      System.out.println("****");
      i++;
    }
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
    int i = 5;
    while (i >= 1) {
      int num = 1;
      for (int j = 1; j <= i; j++) {
        System.out.print(num + "xxx");
        num *= 2;
      }

      System.out.println();
      i--;
    }
  }
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    int i = 1;
    do {
      int num = 1;
      for (int j = 1; j <= i; j++) {
        System.out.print(num + "G");
        num += 2;
      }

      System.out.println();
      i++;
    } while (i <= 5);
  }
}
```

(d)

5.10 Minimizing Numeric Errors

Key Point

Using floating-point numbers in the loop continuation condition may cause numeric errors.

VideoNote

Minimize numeric errors

Numeric errors involving floating-point numbers are inevitable, because floating-point numbers are represented in approximation in computers by nature. This section discusses how to minimize such errors through an example.

Listing 5.8 presents an example summing a series that starts with `0.01` and ends with `1.0`. The numbers in the series will increment by `0.01`, as follows: `0.01 + 0.02 + 0.03`, and so on.

Listing 5.8 TestSum.java

loop

```
public class TestSum {
  public static void main(String[] args) {
    // Initialize sum
    float sum = 0;

    // Add 0.01, 0.02, ..., 0.99, 1 to sum
    for (float i = 0.01f; i <= 1.0f; i = i + 0.01f)
      sum += i;

    // Display result
11       System.out.println("The sum is " + sum);
12     }
13   }
```

```
The sum is 50.499985
```

The `for` loop (lines 7 and 8) repeatedly adds the control variable `i` to `sum`. This variable, which begins with `0.01`, is incremented by `0.01` after each iteration. The loop terminates when `i` exceeds `1.0`.

The `for` loop initial action can be any statement, but it is often used to initialize a control variable. From this example, you can see a control variable can be a `float` type. In fact, it can be any data type.

double precision

The exact `sum` should be `50.50`, but the answer is `50.499985`. The result is imprecise because computers use a fixed number of bits to represent floating-point numbers, and thus they cannot represent some floating-point numbers exactly. If you change `float` in the program to `double`, as follows, you should see a slight improvement in precision, because a `double` variable holds 64 bits, whereas a `float` variable holds 32 bits.

```
// Initialize sum
double sum = 0;

// Add 0.01, 0.02, ..., 0.99, 1 to sum
for (double i = 0.01; i <= 1.0; i = i + 0.01)
  sum += i;
```

numeric error

However, you will be stunned to see the result is actually `49.50000000000003`. What went wrong? If you display `i` for each iteration in the loop, you will see that the last `i` is slightly larger than `1` (not exactly `1`). This causes the last `i` not to be added into `sum`. The fundamental problem is the floating-point numbers are represented by approximation. To fix the problem, use an integer count to ensure all the numbers are added to `sum`. Here is the new loop:

```
double currentValue = 0.01;

for (int count = 0; count < 100; count++) {
  sum += currentValue;
  currentValue += 0.01;
}
```

After this loop, `sum` is `50.50000000000003`. This loop adds the numbers from smallest to biggest. What happens if you add numbers from biggest to smallest (i.e., `1.0`, `0.99`, `0.98`,. . . , `0.02`, `0.01` in this order) is as follows:

```
double currentValue = 1.0;

for (int count = 0; count < 100; count++) {
  sum += currentValue;
  currentValue -= 0.01;
}
```

avoiding numeric error

After this loop, `sum` is `50.49999999999995`. Adding from biggest to smallest is less accurate than adding from smallest to biggest. This phenomenon is an artifact of the finite-precision arithmetic. Adding a very small number to a very big number can have no effect if the result requires more precision than the variable can store. For example, the inaccurate result of `100000000.0 + 0.000000001` is `100000000.0`. To obtain more accurate results, carefully select the order of computation. Adding smaller numbers before bigger numbers to sum is one way to minimize errors.

5.11 Case Studies

Key Point

Loops are fundamental in programming. The ability to write loops is essential in learning Java programming.

If you can write programs using loops, you know how to program! For this reason, this section presents three additional examples of solving problems using loops.

5.11.1 Case Study: Finding the Greatest Common Divisor

gcd

think before you code

The greatest common divisor (gcd) of the two integers **4** and **2** is **2**. The greatest common divisor of the two integers **16** and **24** is **8**. How would you write this program to find the greatest common divisor? Would you immediately begin to write the code? No. It is important to *think before you code*. Thinking enables you to generate a logical solution for the problem without concern about how to write the code.

Let the two input integers be **n1** and **n2**. You know that number **1** is a common divisor, but it may not be the greatest common divisor. Therefore, you can check whether **k** (for **k** = **2**, **3**, **4**, and so on) is a common divisor for **n1** and **n2**, until **k** is greater than **n1** or **n2**. Store the common divisor in a variable named **gcd**. Initially, **gcd** is **1**. Whenever a new common divisor is found, it becomes the new **gcd**. When you have checked all the possible common divisors from **2** up to **n1** or **n2**, the value in variable **gcd** is the greatest common divisor.

logical solution

Once you have a *logical solution*, type the code to translate the solution into a Java program as follows:

```
int gcd = 1; // Initial gcd is 1
int k = 2; // Possible gcd

while (k <= n1 && k <= n2) {
  if (n1 % k == 0 && n2 % k == 0)
    gcd = k; // Update gcd
  k++; // Next possible gcd
}

// After the loop, gcd is the greatest common divisor for n1 and n2
```

Listing 5.9 presents the program that prompts the user to enter two positive integers and finds their greatest common divisor.

LISTING 5.9 `GreatestCommonDivisor.java`

```
import java.util.Scanner;

public class GreatestCommonDivisor {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter first integer: ");
    int n1 = input.nextInt();
    System.out.print("Enter second integer: ");
    int n2 = input.nextInt();

    int gcd = 1; // Initial gcd is 1
    int k = 2; // Possible gcd
    while (k <= n1 && k <= n2) {
      if (n1 % k == 0 && n2 % k == 0)
        gcd = k; // Update gcd
      k++;
```

input

input

gcd

check divisor

```
      }

      System.out.println("The greatest common divisor for " + n1 +
        " and " + n2 + " is " + gcd);
    }
  }
```

output

```
Enter first integer: 125 ↵Enter
Enter second integer: 2525 ↵Enter
The greatest common divisor for 125 and 2525 is 25
```

Translating a logical solution to Java code is not unique. For example, you could use a `for` loop to rewrite the code as follows:

think before you type

```
for (int k = 2; k <= n1 && k <= n2; k++) {
  if (n1 % k == 0 && n2 % k == 0)
    gcd = k;
}
```

A problem often has multiple solutions, and the gcd problem can be solved in many ways. Programming Exercise 5.14 suggests another solution. A more efficient solution is to use the classic Euclidean algorithm (see Section 22.6).

multiple solutions

You might think that a divisor for a number `n1` cannot be greater than `n1 / 2` and would attempt to improve the program using the following loop:

erroneous solutions

```
for (int k = 2; k <= n1 / 2 && k <= n2 / 2; k++) {
  if (n1 % k == 0 && n2 % k == 0)
    gcd = k;
}
```

This revision is wrong. Can you find the reason? See Checkpoint Question 5.11.1 for the answer.

5.11.2 Case Study: Predicting the Future Tuition

Suppose the tuition for a university is `$10,000` this year and tuition increases `7%` every year. In how many years will the tuition be doubled?

Before you can write a program to solve this problem, first consider how to solve it by hand. The tuition for the second year is the tuition for the first year * `1.07`. The tuition for a future year is the tuition of its preceding year * `1.07`. Thus, the tuition for each year can be computed as follows:

think before you code

```
double tuition = 10000; int year = 0; // Year 0
tuition = tuition * 1.07; year++;     // Year 1
tuition = tuition * 1.07; year++;     // Year 2
tuition = tuition * 1.07; year++;     // Year 3
...
```

Keep computing the tuition for a new year until it is at least `20000`. By then, you will know how many years it will take for the tuition to be doubled. You can now translate the logic into the following loop:

```
double tuition = 10000; // Year 0
int year = 0;
while (tuition < 20000) {
  tuition = tuition * 1.07;
  year++;
}
```

The complete program is given in Listing 5.10.

LISTING 5.10 `FutureTuition.java`

loop
next year's tuition

```
 1  public class FutureTuition {
 2    public static void main(String[] args) {
 3      double tuition = 10000; // Year 0
 4      int year = 0;
 5      while (tuition < 20000) {
 6        tuition = tuition * 1.07;
 7        year++;
 8      }
 9
10      System.out.println("Tuition will be doubled in "
11        + year + " years");
12      System.out.printf("Tuition will be $%.2f in %1d years",
13        tuition, year);
14    }
15  }
```

```
Tuition will be doubled in 11 years
Tuition will be $21048.52 in 11 years
```

The `while` loop (lines 5–8) is used to repeatedly compute the tuition for a new year. The loop terminates when the tuition is greater than or equal to `20000`.

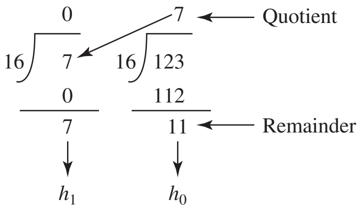
5.11.3 Case Study: Converting Decimals to Hexadecimals

Hexadecimals are often used in computer systems programming (see Appendix F for an introduction to number systems). How do you convert a decimal number to a hexadecimal number? To convert a decimal number *d* to a hexadecimal number is to find the hexadecimal digits $h_n, h_{n-1}, h_{n-2}, \ldots, h_2, h_1$, and h_0 such that

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \cdots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

These hexadecimal digits can be found by successively dividing *d* by 16 until the quotient is 0. The remainders are $h_0, h_1, h_2, \ldots, h_{n-2}, h_{n-1}$, and h_n. The hexadecimal digits include the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, plus A, which is the decimal value 10; B, which is the decimal value 11; C, which is 12; D, which is 13; E, which is 14; and F, which is 15.

For example, the decimal number `123` is `7B` in hexadecimal. The conversion is done as follows. Divide `123` by `16`. The remainder is `11` (`B` in hexadecimal) and the quotient is `7`. Continue to divide `7` by `16`. The remainder is `7` and the quotient is `0`. Therefore, `7B` is the hexadecimal number for `123`.



Listing 5.11 gives a program that prompts the user to enter a decimal number and converts it into a hex number as a string.

LISTING 5.11 Dec2Hex.java

```
 1  import java.util.Scanner;
 2
 3  public class Dec2Hex {
 4    /** Main method */
 5    public static void main(String[] args) {
 6      // Create a Scanner
 7      Scanner input = new Scanner(System.in);
 8
 9      // Prompt the user to enter a decimal integer
10      System.out.print("Enter a decimal number: ");
11      int decimal = input.nextInt();
12
13      // Convert decimal to hex
14      String hex = "";
15
16      while (decimal != 0) {
17        int hexValue = decimal % 16;
18
19        // Convert a decimal value to a hex digit
20        char hexDigit = (0 <= hexValue && hexValue <= 9)?
21          (char)(hexValue + '0'): (char)(hexValue - 10 + 'A');
22
23        hex = hexDigit + hex;
24        decimal = decimal / 16;
25      }
26
27      System.out.println("The hex number is " + hex);
28    }
29  }
```

input decimal

decimal to hex

get a hex char

add to hex string

```
Enter a decimal number: 1234 ↵Enter
The hex number is 4D2
```

	line#	decimal	hex	hexValue	hexDigit
	14	1234	""		
iteration 1	17			2	
	23		"2"		2
	24	77			
iteration 2	17			13	
	23		"D2"		D
	24	4			
iteration 3	17			4	
	23		"4D2"		4
	24	0			

The program prompts the user to enter a decimal integer (line 11), converts it to a hex number as a string (lines 14–25), and displays the result (line 27). To convert a decimal to a hex number, the program uses a loop to successively divide the decimal number by **16** and obtain its remainder (line 17). The remainder is converted into a hex character (lines 20 and 21). The character is then appended to the hex string (line 23). The hex string is initially empty (line 14). Divide the decimal number by **16** to remove a hex digit from the number (line 24). The loop ends when the remaining decimal number becomes **0**.

The program converts a `hexValue` between `0` and `15` into a hex character. If `hexValue` is between `0` and `9`, it is converted to `(char)(hexValue +'0')` (line 21). Recall that when adding a character with an integer, the character's Unicode is used in the evaluation. For example, if `hexValue` is `5`, `(char)(hexValue + '0')` returns `5`. Similarly, if `hexValue` is between `10` and `15`, it is converted to `(char)(hexValue – 10 + 'A')` (line 21). For instance, if `hexValue` is `11`, `(char),(hexValue – 10 + 'A')` returns `B`.

Check Point

5.11.1 Will the program work if `n1` and `n2` are replaced by `n1 / 2` and `n2 / 2` in line 17 in Listing 5.9?

5.11.2 In Listing 5.11, why is it wrong if you change the code `(char)(hexValue + '0')` to `hexValue + '0'` in line 21?

5.11.3 In Listing 5.11, how many times the loop body is executed for a decimal number `245`, and how many times the loop body is executed for a decimal number `3245`?

5.11.4 What is the hex number after E? What is the hex number after F?

5.11.5 Revise line 27 in Listing 5.11 so the program displays hex number 0 if the input decimal is 0.

5.12 Keywords *break* and *continue*

Key Point

The `break` and `continue` keywords provide additional controls in a loop.

Pedagogical Note

Two keywords, `break` and `continue`, can be used in loop statements to provide additional controls. Using `break` and `continue` can simplify programming in some cases. Overusing or improperly using them, however, can make programs difficult to read and debug. (*Note to instructors*: You may skip this section without affecting students' understanding of the rest of the book.)

break statement

You have used the keyword `break` in a `switch` statement. You can also use `break` in a loop to immediately terminate the loop. Listing 5.12 presents a program to demonstrate the effect of using `break` in a loop.

Listing 5.12 `TestBreak.java`

break

```
public class TestBreak {
  public static void main(String[] args) {
    int sum = 0;
    int number = 0;

    while (number < 20) {
      number++;
      sum += number;
      if (sum >= 100)
        break;
    }

    System.out.println(“The number is “ + number);
    System.out.println(“The sum is “ + sum);
  }
}
```

```
The number is 14
The sum is 105
```

The program in Listing 5.12 adds integers from `1` to `20` in this order to `sum` until `sum` is greater than or equal to `100`. Without the `if` statement (line 9), the program calculates the sum of the numbers from `1` to `20`. However, with the `if` statement, the loop terminates when `sum` becomes greater than or equal to `100`. Without the `if` statement, the output would be as follows:

```
The number is 20
The sum is 210
```

continue statement

You can also use the `continue` keyword in a loop. When it is encountered, it ends the current iteration and program control goes to the end of the loop body. In other words, `continue` breaks out of an iteration, while the `break` keyword breaks out of a loop. Listing 5.13 presents a program to demonstrate the effect of using `continue` in a loop.

Listing 5.13 TestContinue.java

continue

```
 1  public class TestContinue {
 2    public static void main(String[] args) {
 3      int sum = 0;
 4      int number = 0;
 5
 6      while (number < 20) {
 7        number++;
 8        if (number == 10 || number == 11)
 9          continue;
10        sum += number;
11      }
12
13      System.out.println("The sum is " + sum);
14    }
15  }
```

```
The sum is 189
```

The program in Listing 5.13 adds integers from `1` to `20` except `10` and `11` to `sum`. With the `if` statement in the program (line 8), the `continue` statement is executed when `number` becomes `10` or `11`. The `continue` statement ends the current iteration so that the rest of the statement in the loop body is not executed; therefore, `number` is not added to `sum` when it is `10` or `11`. Without the `if` statement in the program, the output would be as follows:

```
The sum is 210
```

In this case, all of the numbers are added to `sum`, even when `number` is `10` or `11`. Therefore, the result is `210`, which is `21` more than it was with the `if` statement.

Note

The `continue` statement is always inside a loop. In the `while` and `do-while` loops, the `loop-continuation-condition` is evaluated immediately after the `continue` statement. In the `for` loop, the `action-after-each-iteration` is performed, then the `loop-continuation-condition` is evaluated immediately after the `continue` statement.

goto

Note

Some programming languages have a `goto` statement. The `goto` statement indiscriminately transfers control to any statement in the program and executes it. This makes your program vulnerable to errors. The `break` and `continue` statements in Java are different from `goto` statements. They operate only in a loop or a `switch` statement. The `break` statement breaks out of the loop, and the `continue` statement breaks out of the current iteration in the loop.

You can always write a program without using `break` or `continue` in a loop (see CheckPoint Question 5.12.3). In general, though, using `break` and `continue` is appropriate if it simplifies coding and makes programs easier to read.

Suppose you need to write a program to find the smallest factor other than `1` for an integer `n` (assume `n >= 2`). You can write a simple and intuitive code using the `break` statement as follows:

```
int factor = 2;
while (factor <= n) {
  if (n % factor == 0)
    break;
  factor++;
}
System.out.println("The smallest factor other than 1 for "
  + n + " is " + factor);
```

You may rewrite the code without using `break` as follows:

```
boolean found = false;
int factor = 2;
while (factor <= n && !found) {
  if (n % factor == 0)
    found = true;
  else
    factor++;
}
System.out.println("The smallest factor other than 1 for "
  + n + " is " + factor);
```

Obviously, the `break` statement makes this program simpler and easier to read in this case. However, you should use `break` and `continue` with caution. Too many `break` and `continue` statements will produce a loop with many exit points and make the program difficult to read.

Note

Programming is a creative endeavor. There are many different ways to write code. In fact, you can find a smallest factor using a rather simple code as follows:

```
int factor = 2;
while (n % factor != 0)
  factor++;
or
for (int factor = 2; n % factor != 0; factor++);
```

The code here finds the smallest factor for an integer `n`. Programming Exercise 5.16 writes a program that finds all smallest factors in `n`.

Check
Point

5.12.1 What is the keyword `break` for? What is the keyword `continue` for? Will the following programs terminate? If so, give the output.

```
int balance = 10;
while (true) {
  if (balance < 9)
    break;
  balance = balance - 9;
}

System.out.println("Balance is "
  + balance);
```

(a)

```
int balance = 10;
while (true) {
  if (balance < 9)
    continue;
  balance = balance - 9;
}

System.out.println("Balance is "
  + balance);
```

(b)

5.12.2 The `for` loop on the left is converted into the `while` loop on the right. What is wrong? Correct it.

```
int sum = 0;
for (int i = 0; i < 4; i++) {
  if (i % 3 == 0) continue;
  sum += i;
}
```

Converted

Wrong conversion

```
int i = 0, sum = 0;
while (i < 4) {
  if (i % 3 == 0) continue;
  sum += i;
  i++;
}
```

5.12.3 Rewrite the programs `TestBreak` and `TestContinue` in Listings 5.12 and 5.13 without using `break` and `continue`.

5.12.4 After the `break` statement in (a) is executed in the following loop, which statement is executed? Show the output. After the `continue` statement in (b) is executed in the following loop, which statement is executed? Show the output.

```
for (int i = 1; i < 4; i++) {
  for (int j = 1; j < 4; j++) {
    if (i * j > 2)
      break;

    System.out.println(i * j);
  }

  System.out.println(i);
}
```

(a)

```
for (int i = 1; i < 4; i++) {
  for (int j = 1; j < 4; j++) {
    if (i * j > 2)
      continue;

    System.out.println(i * j);
  }

  System.out.println(i);
}
```

(b)

5.13 Case Study: Checking Palindromes

This section presents a program that checks whether a string is a palindrome.

Key Point

A string is a palindrome if it reads the same forward and backward. The words “mom,” “dad,” and “noon,” for instance, are all palindromes.

The problem is to write a program that prompts the user to enter a string and reports whether the string is a palindrome. One solution is to check whether the first character in the string is the same as the last character. If so, check whether the second character is the same as the second-to-last think before you code

character. This process continues until a mismatch is found or all the characters in the string are checked, except for the middle character if the string has an odd number of characters.

Listing 5.14 gives the program.

LISTING 5.14 Palindrome.java

input string

low index

high index

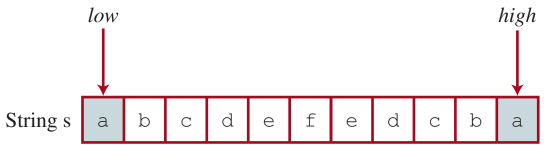
update indices

```
 1  import java.util.Scanner;
 2
 3  public class Palindrome {
 4    /** Main method */
 5    public static void main(String[] args) {
 6      // Create a Scanner
 7      Scanner input = new Scanner(System.in);
 8
 9      // Prompt the user to enter a string
10      System.out.print("Enter a string: ");
11      String s = input.nextLine();
12
13      // The index of the first character in the string
14      int low = 0;
15
16      // The index of the last character in the string
17      int high = s.length() - 1;
18
19      boolean isPalindrome = true;
20      while (low < high) {
21        if (s.charAt(low) != s.charAt(high)) {
22          isPalindrome = false;
23          break;
24        }
25
26        low++;
27        high--;
28      }
29
30      if (isPalindrome)
31        System.out.println(s + " is a palindrome");
32      else
33        System.out.println(s + " is not a palindrome");
34    }
35  }
```

```
Enter a string: noon ↵ Enter
noon is a palindrome
```

```
Enter a string: abcdefgnhgfedcba ↵ Enter
abcdefgnhgfedcba is not a palindrome
```

The program uses two variables, `low` and `high`, to denote the positions of the two characters at the beginning and the end in a string `s` (lines 14 and 17), as shown in the following figure.



Initially, `low` is `0` and `high` is `s.length() – 1`. If the two characters at these positions match, increment `low` by `1` and decrement `high` by `1` (lines 26–27). This process continues until (`low >= high`) or a mismatch is found (line 21).

The program uses a `boolean` variable `isPalindrome` to denote whether the string `s` is a palindrome. Initially, it is set to `true` (line 19). When a mismatch is discovered (line 21), `isPalindrome` is set to `false` (line 22) and the loop is terminated with a break statement (line 23).

Check Point

5.13.1 What happens to the program if (`low < high`) in line 20 is changed to (`low <= high`)?

5.14 Case Study: Displaying Prime Numbers

Key Point

This section presents a program that displays the first 50 prime numbers in 5 lines, each containing 10 numbers.

An integer greater than `1` is *prime* if its only positive divisor is `1` or itself. For example, `2`, `3`, `5`, and `7` are prime numbers, but `4`, `6`, `8`, and `9` are not.

The problem is to display the first 50 prime numbers in 5 lines, each of which contains 10 numbers. The problem can be broken into the following tasks:

- Determine whether a given number is prime.
- For `number` = `2`, `3`, `4`, `5`, `6`, …, test whether it is prime.
- Count the prime numbers.
- Display each prime number and display 10 numbers per line.

Obviously, you need to write a loop and repeatedly test whether a new `number` is prime. If the `number` is prime, increase the count by `1`. The `count` is `0` initially. When it reaches `50`, the loop terminates.

Here is the algorithm for the problem:

```
Set the number of prime numbers to be printed as
  a constant NUMBER_OF_PRIMES;
Use count to track the number of prime numbers and
  set an initial count to 0;
Set an initial number to 2;

while (count < NUMBER_OF_PRIMES) {
  Test whether number is prime;

  if number is prime {
    Display the prime number and increase the count;
  }

  Increment number by 1;
}
```

To test whether a number is prime, check whether it is divisible by `2`, `3`, `4`, and so on up to `number/2`. If a divisor is found, the number is not a prime. The algorithm can be described as follows:

```
Use a boolean variable isPrime to denote whether
  the number is prime; Set isPrime to true initially;

for (int divisor = 2; divisor <= number / 2; divisor++) {
  if (number % divisor == 0) {
    Set isPrime to false
    Exit the loop;
  }
}
```

The complete program is given in Listing 5.15.

Listing 5.15 `PrimeNumber.java`

count prime numbers

check primeness

exit loop

display if prime

```
 1  public class PrimeNumber {
 2    public static void main(String[] args) {
 3      final int NUMBER_OF_PRIMES = 50; // Number of primes to display
 4      final int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
 5      int count = 0; // Count the number of prime numbers
 6      int number = 2; // A number to be tested for primeness
 7
 8      System.out.println("The first 50 prime numbers are \n");
 9
10      // Repeatedly find prime numbers
11      while (count < NUMBER_OF_PRIMES) {
12        // Assume the number is prime
13        boolean  isPrime = true; // Is the current number prime?
14
15        // Test whether number is prime
16        for (int divisor = 2; divisor <= number / 2; divisor++) {
17          if (number % divisor == 0) {  // If true, number is not prime
18            isPrime = false; // Set isPrime to false
19            break; // Exit the for loop
20          }
21        }
22
23        // Display the prime number and increase the count
24        if (isPrime) {
25          count++; // Increase the count
26
27          if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
28            // Display the number and advance to the new line
29            System.out.println(number);
30          }
31          else
32            System.out.print(number + " ");
33        }
34
35        // Check if the next number is prime
36        number++;
37      }
38    }
39  }
```

```
The first 50 prime numbers are
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
```

subproblem

This is a complex program for novice programmers. The key to developing a programmatic solution for this problem, and for many other problems, is to break it into subproblems and develop solutions for each of them in turn. Do not attempt to develop a complete solution in the first trial. Instead, begin by writing the code to determine whether a given number is prime, then expand the program to test whether other numbers are prime in a loop.

To determine whether a number is prime, check whether it is divisible by a number between `2` and `number/2` inclusive (lines 16–21). If so, it is not a prime number (line 18); otherwise, it is a prime number. For a prime number, display it (lines 27–33). If the count is divisible by `10`, display the number followed by a newline (lines 27–30). The program ends when the count reaches `50`.

The program uses the `break` statement in line 19 to exit the `for` loop as soon as the number is found to be a nonprime. You can rewrite the loop (lines 16–21) without using the `break` statement, as follows:

```
for (int divisor = 2; divisor <= number / 2 && isPrime;
     divisor++) {
  // If true, the number is not prime
  if (number % divisor == 0) {
    // Set isPrime to false, if the number is not prime
    isPrime = false;
  }
}
```

However, using the `break` statement makes the program simpler and easier to read in this case.

Prime numbers have many applications in computer science. Section 22.7 will study several efficient algorithms for finding prime numbers.

Check Point

5.14.1 Simplify the code in lines 27–32 using a conditional operator.

KEY TERMS

`break` statement	186	loop body	160
`continue` statement	187	nested loop	178
`do-while` loop	171	off-by-one error	162
`for` loop	173	output redirection	170
infinite loop	162	posttest loop	176
input redirection	170	pretest loop	176
iteration	160	sentinel value	168
loop	160	`while` loop	160

CHAPTER SUMMARY

1. There are three types of repetition statements: the `while` loop, the `do-while` loop, and the `for` loop.
2. The part of the loop that contains the statements to be repeated is called the *loop body*.
3. A one-time execution of a loop body is referred to as an *iteration of the loop*.
4. An *infinite loop* is a loop statement that executes infinitely.
5. In designing loops, you need to consider both the *loop control structure* and the loop body.
6. The `while` loop checks the `loop-continuation-condition` first. If the condition is `true`, the loop body is executed; if it is `false`, the loop terminates.
7. The `do-while` loop is similar to the `while` loop, except the `do-while` loop executes the loop body first then checks the `loop-continuation-condition` to decide whether to continue or to terminate.
8. The `while` loop and the `do-while` loop often are used when the number of repetitions is not predetermined.

9. A *sentinel value* is a special value that signifies the end of the loop.
10. The `for` loop generally is used to execute a loop body a fixed number of times.
11. The `for` loop control has three parts. The first part is an initial action that often initializes a control variable. The second part, the `loop-continuation-condition`, determines whether the loop body is to be executed. The third part is executed after each iteration and is often used to adjust the control variable. Usually, the loop control variables are initialized and changed in the control structure.
12. The `while` loop and `for` loop are called *pretest loops* because the continuation condition is checked before the loop body is executed.
13. The `do-while` loop is called a *posttest loop* because the condition is checked after the loop body is executed.
14. Two keywords `break` and `continue` can be used in a loop.
15. The `break` keyword immediately ends the innermost loop, which contains the break.
16. The `continue` keyword only ends the current iteration.

QUIZ

Answer the quiz for this chapter online at the Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

read and think before coding

explore solutions

Pedagogical Note

Read each problem several times until you understand it. Think how to solve the problem before starting to write code. Translate your logic into a program.

A problem often can be solved in many different ways. Students are encouraged to explore various solutions.

Sections 5.2–5.7

***5.1** (*Count positive and negative numbers and compute the average of numbers*) Write a program that reads an unspecified number of integers, determines how many positive and negative values have been read, and computes the total and average of the input values (not counting zeros). Your program ends with the input `0`. Display the average as a floating-point number. Here are sample runs:

```
Enter an integer, the input ends if it is 0: 1 2 -1 3 0 ↵Enter
The number of positives is 3
The number of negatives is 1
The total is 5.0
The average is 1.25
```

```
Enter an integer, the input ends if it is 0: 0 ↵Enter
No numbers are entered except 0
```

5.2 (*Repeat additions*) Listing 5.4, SubtractionQuizLoop.java, generates five random subtraction questions. Revise the program to generate 10 random addition questions for two integers between `1` and `15`. Display the correct count and test time.

5.3 (*Conversion from kilograms to pounds*) Write a program that displays the following table (note `1` kilogram is `2.2` pounds):

```
Kilograms    Pounds
1               2.2
3               6.6
. . .
197           433.4
199           437.8
```

5.4 (*Conversion from miles to kilometers*) Write a program that displays the following table (note `1` mile is `1.609` kilometers):

```
Miles       Kilometers
1           1.609
2           3.218
. . .
9           14.481
10          16.090
```

5.5 (*Conversion from kilograms to pounds and pounds to kilograms*) Write a program that displays the following two tables side by side:

```
Kilograms       Pounds    |   Pounds        Kilograms
1                 2.2     |   20              9.09
3                 6.6     |   25             11.36
. . .
197             433.4     |   510           231.82
199             437.8     |   515           234.09
```

5.6 (*Conversion from miles to kilometers*) Write a program that displays the following two tables side by side:

```
Miles       Kilometers     |   Kilometers   Miles
1           1.609          |   20           12.430
2           3.218          |   25           15.538
. . .
9           14.481         |   60           37.290
10          16.090         |   65           40.398
```

****5.7** (*Financial application: compute future tuition*) Suppose the tuition for a university is $10,000 this year and increases 5% every year. In one year, the tuition will be $10,500. Write a program that displays the tuition in 10 years, and the total cost of four years' worth of tuition starting after the tenth year.

5.8 (*Find the highest score*) Write a program that prompts the user to enter the number of students and each student's name and score, and finally displays the name of the student with the highest score. Use the `next()` method in the `Scanner` class to read a name, rather than using the `nextLine()` method. Assume that the number of students is at least 1.

***5.9** (*Find the two highest scores*) Write a program that prompts the user to enter the number of students and each student's name and score, and finally displays the student with the highest score and the student with the second-highest score. Use the `next()` method in the `Scanner` class to read a name rather than using the `nextLine()` method. Assume that the number of students is at least 2.

5.10 (*Find numbers divisible by 5 and 6*) Write a program that displays all the numbers from 100 to 1,000 (10 per line) that are divisible by 5 and 6. Numbers are separated by exactly one space.

5.11 (*Find numbers divisible by 5 or 6, but not both*) Write a program that displays all the numbers from 100 to 200 (10 per line) that are divisible by 5 or 6, but not both. Numbers are separated by exactly one space.

5.12 (*Find the smallest* n such that $n^2 > 12{,}000$) Use a `while` loop to find the smallest integer n such that n^2 is greater than 12,000.

5.13 (*Find the largest* n such that $n^3 < 12{,}000$) Use a `while` loop to find the largest integer n such that n^3 is less than 12,000.

Sections 5.8–5.10

***5.14** (*Compute the greatest common divisor*) Another solution for Listing 5.9 to find the greatest common divisor of two integers `n1` and `n2` is as follows: First find `d` to be the minimum of `n1` and `n2`, then check whether `d`, `d–1`, `d–2`, …, `2`, or `1` is a divisor for both `n1` and `n2` in this order. The first such common divisor is the greatest common divisor for `n1` and `n2`. Write a program that prompts the user to enter two positive integers and displays the gcd.

***5.15** (*Display the ASCII character table*) Write a program that prints the characters in the ASCII character table from `!` to `~`. Display 10 characters per line. The ASCII table is given in Appendix B. Characters are separated by exactly one space.

***5.16** (*Find the factors of an integer*) Write a program that reads an integer and displays all its smallest factors in an increasing order. For example, if the input integer is `120`, the output should be as follows: `2`, `2`, `2`, `3`, `5`.

****5.17** (*Display pyramid*) Write a program that prompts the user to enter an integer from `1` to `15` and displays a pyramid, as presented in the following sample run:

```
Enter the number of lines: 7 ↵Enter
                  1
               2  1  2
            3  2  1  2  3
         4  3  2  1  2  3  4
      5  4  3  2  1  2  3  4  5
   6  5  4  3  2  1  2  3  4  5  6
7  6  5  4  3  2  1  2  3  4  5  6  7
```

***5.18** (*Display four patterns using loops*) Use nested loops that display the following patterns in four separate programs:

```
Pattern A        Pattern B       Pattern C       Pattern D
1                1 2 3 4 5 6               1     1 2 3 4 5 6
1 2              1 2 3 4 5               2 1       1 2 3 4 5
1 2 3            1 2 3 4               3 2 1         1 2 3 4
1 2 3 4          1 2 3               4 3 2 1           1 2 3
1 2 3 4 5        1 2               5 4 3 2 1             1 2
1 2 3 4 5 6      1               6 5 4 3 2 1               1
```

****5.19** (*Display numbers in a pyramid pattern*) Write a nested `for` loop that prints the following output:

```
                               1
                           1   2   1
                       1   2   4   2   1
                   1   2   4   8   4   2   1
               1   2   4   8  16   8   4   2   1
           1   2   4   8  16  32  16   8   4   2   1
       1   2   4   8  16  32  64  32  16   8   4   2   1
   1   2   4   8  16  32  64 128  64  32  16   8   4   2   1
```

***5.20** (*Display prime numbers between 2 and 1,000*) Modify the program given in Listing 5.15 to display all the prime numbers between 2 and 1,000, inclusive. Display eight prime numbers per line. Numbers are separated by exactly one space.

Comprehensive

****5.21** (*Financial application: compare loans with various interest rates*) Write a program that lets the user enter the loan amount and loan period in number of years, and displays the monthly and total payments for each interest rate starting from 5% to 8%, with an increment of 1/8. Here is a sample run:

```
Loan Amount: 10000 ↵Enter
Number of Years: 5 ↵Enter

Interest Rate    Monthly Payment    Total Payment
5.000%           188.71             11322.74
5.125%           189.29             11357.13
5.250%           189.86             11391.59
...
7.875%           202.17             12129.97
8.000%           202.76             12165.84
```

For the formula to compute monthly payment, see Listing 2.9, ComputeLoan.java.

****5.22** (*Financial application: loan amortization schedule*) The monthly payment for a given loan pays the principal and the interest. The monthly interest is computed by multiplying the monthly interest rate and the balance (the remaining principal). The principal paid for the month is therefore the monthly payment minus the monthly interest. Write a program that lets the user enter the loan amount, number of years, and interest rate then displays the amortization schedule for the loan. Here is a sample run:

VideoNote
Display loan schedule

```
Loan Amount: 10000 ↵Enter
Number of Years: 1 ↵Enter
Annual Interest Rate: 7 ↵Enter

Monthly Payment: 865.26
Total Payment: 10383.21

Payment#      Interest     Principal    Balance
1                58.33        806.93     9193.07
2                53.62        811.64     8381.43
...
11               10.00        855.26      860.27
12                5.01        860.25        0.01
```

Note

The balance after the last payment may not be zero. If so, the last payment should be the normal monthly payment plus the final balance.

Hint: Write a loop to display the table. Since the monthly payment is the same for each month, it should be computed before the loop. The balance is initially the loan amount. For each iteration in the loop, compute the interest and principal, and update the balance. The loop may look as follows:

```
for (i = 1; i <= numberOfYears * 12; i++) {
  interest = monthlyInterestRate * balance;
  principal = monthlyPayment - interest;
  balance = balance - principal;
  System.out.println(i + "\t\t" + interest
    + "\t\t" + principal + "\t\t" + balance);
}
```

***5.23** (*Demonstrate cancellation errors*) A cancellation error occurs when you are manipulating a very large number with a very small number. The large number may cancel out the smaller number. For example, the result of `100000000.0 + 0.000000001` is equal to `100000000.0`. To avoid cancellation errors and obtain more accurate results, carefully select the order of computation. For example, in computing the following summation, you will obtain more accurate results by computing from right to left rather than from left to right:

$$1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

Write a program that compares the results of the summation of the preceding series, computing from left to right and from right to left with `n = 50000`.

VideoNote

Sum a series

***5.24** (*Sum a series*) Write a program to compute the following summation:

$$\frac{1}{3} + \frac{3}{5} + \frac{5}{7} + \frac{7}{9} + \frac{9}{11} + \frac{11}{13} + \cdots + \frac{95}{97} + \frac{97}{99}$$

****5.25** (*Compute π*) You can approximate π by using the following summation:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots + \frac{(-)^{i+1}}{2i - 1}\right)$$

Write a program that displays the π value for `i = 10000`, `20000`, …, and `100000`.

****5.26** (*Compute e*) You can approximate `e` using the following summation:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots + \frac{1}{i!}$$

Write a program that displays the `e` value for `i = 1`, `2`, …, and `20`. Format the number to display 16 digits after the decimal point. (*Hint:* Because $i! = i \times (i - 1) \times \ldots \times 2 \times 1$, then

$$\frac{1}{i!} \text{ is } \frac{1}{i(i-1)!}$$

Initialize `e` and `item` to be `1`, and keep adding a new `item` to `e`. The new item is the previous item divided by `i`, for `i >= 2`.)

****5.27** (*Display leap years*) Write a program that displays all the leap years, 10 per line, from 101 to 2100, separated by exactly one space. Also display the number of leap years in this period.

****5.28** (*Display the first days of each month*) Write a program that prompts the user to enter the year and first day of the year, then displays the first day of each month in the year. For example, if the user entered the year `2013`, and `2` for Tuesday, January 1, 2013, your program should display the following output:

```
January 1, 2013 is Tuesday
...
December 1, 2013 is Sunday
```

****5.29** (*Display calendars*) Write a program that prompts the user to enter the year and first day of the year and displays the calendar table for the year on the console. For example, if the user entered the year `2013`, and `2` for Tuesday, January 1, 2013, your program should display the calendar for each month in the year, as follows:

```
                  January 2013
 ---------------------------------------------
  Sun   Mon   Tue   Wed   Thu   Fri   Sat
                1     2     3     4     5
    6     7     8     9    10    11    12
   13    14    15    16    17    18    19
   20    21    22    23    24    25    26
   27    28    29    30    31

                    ...

                  December 2013
 ---------------------------------------------
  Sun   Mon   Tue   Wed   Thu   Fri   Sat
    1     2     3     4     5     6     7
    8     9    10    11    12    13    14
   15    16    17    18    19    20    21
   22    23    24    25    26    27    28
   29    30    31
```

***5.30** (*Financial application: compound value*) Suppose you save $100 *each* month into a savings account with the annual interest rate 5%. Thus, the monthly interest rate is `0.05 / 12 = 0.00417`. After the first month, the value in the account becomes

```
100 * (1 + 0.00417) = 100.417
```

After the second month, the value in the account becomes

```
(100 + 100.417) * (1 + 0.00417) = 201.252
```

After the third month, the value in the account becomes

```
(100 + 201.252) * (1 + 0.00417) = 302.507
```

and so on.

Write a program that prompts the user to enter an amount (e.g., `100`), the annual interest rate (e.g., `5`), and the number of months (e.g., `6`) then displays the amount in the savings account after the given month.

***5.31** (*Financial application: compute CD value*) Suppose you put $10,000 into a CD with an annual percentage yield of 5.75%. After one month, the CD is worth

```
10000 + 10000 * 5.75 / 1200 = 10047.92
```

After two months, the CD is worth

```
10047.91 + 10047.91 * 5.75 / 1200 = 10096.06
```

After three months, the CD is worth

```
10096.06 + 10096.06 * 5.75 / 1200 = 10144.44
```

and so on.

Write a program that prompts the user to enter an amount (e.g., `10000`), the annual percentage yield (e.g., `5.75`), and the number of months (e.g., `18`) and displays a table as presented in the sample run.

```
Enter the initial deposit amount: 10000 ↵Enter
Enter annual percentage yield: 5.75 ↵Enter
Enter maturity period (number of months): 18 ↵Enter

Month    CD Value
1        10047.92
2        10096.06
...
17       10846.57
18       10898.54
```

****5.32** (*Game: lottery*) Revise Listing 3.8, Lottery.java, to generate a lottery of a two-digit number. The two digits in the number are distinct. (*Hint*: Generate the first digit. Use a loop to continuously generate the second digit until it is different from the first digit.)

****5.33** (*Perfect number*) A positive integer is called a *perfect number* if it is equal to the sum of all of its positive divisors, excluding itself. For example, 6 is the first perfect number because `6 = 3 + 2 + 1`. The next is `28 = 14 + 7 + 4 + 2 + 1`. There are four perfect numbers $< 10{,}000$. Write a program to find all these four numbers.

*****5.34** (*Game: scissor, rock, paper*) Programming Exercise 3.17 gives a program that plays the scissor–rock–paper game. Revise the program to let the user continuously play until either the user or the computer wins more than two times than its opponent.

***5.35** (*Summation*) Write a program to compute the following summation:

$$\frac{1}{1+\sqrt{2}} + \frac{1}{\sqrt{2}+\sqrt{3}} + \frac{1}{\sqrt{3}+\sqrt{4}} + \ldots + \frac{1}{\sqrt{624}+\sqrt{625}}$$

****5.36** (*Business application: checking ISBN*) Use loops to simplify Programming Exercise 3.9.

****5.37** (*Decimal to binary*) Write a program that prompts the user to enter a decimal integer then displays its corresponding binary value. Don't use Java's `Integer.toBinaryString(int)` in this program.

****5.38** (*Decimal to octal*) Write a program that prompts the user to enter a decimal integer and displays its corresponding octal value. Don't use Java's `Integer.toOctalString(int)` in this program.

***5.39** (*Financial application: find the sales amount*) You have just started a sales job in a department store. Your pay consists of a base salary and a commission. The base salary is $5,000. The scheme shown below is used to determine the commission rate.

Sales Amount	Commission Rate
`$0.01–$5,000`	`8%`
`$5,000.01–$10,000`	`10%`
`$10,000.01 and above`	`12%`

Note this is a graduated rate. The rate for the first $5,000 is at 8%, the next $5,000 is at 10%, and the rest is at 12%. If the sales amount is 25,000, the commission is `5,000 * 8 + 5,000 * 10 + 15,000 * 12 = 2,700`
Your goal is to earn $30,000 a year. Write a program that finds out the minimum number of sales you have to generate in order to make $30,000.

5.40 (*Simulation: heads or tails*) Write a program that simulates flipping a coin one million times and displays the number of heads and tails.

***5.41** (*Occurrence of max numbers*) Write a program that reads integers, finds the largest of them, and counts its occurrences. Assume the input ends with number `0`. Suppose you entered `3 5 2 5 5 5 0`; the program finds that the largest is `5` and the occurrence count for `5` is `4`. If no input is entered, display "No numbers are entered except 0".

(*Hint:* Maintain two variables, `max` and `count`. `max` stores the current max number and `count` stores its occurrences. Initially, assign the first number to `max` and `1` to `count`. Compare each subsequent number with `max`. If the number is greater than `max`, assign it to `max` and reset `count` to `1`. If the number is equal to `max`, increment `count` by `1`.)

```
Enter numbers: 3 5 2 5 5 5 0 ↵Enter
The largest number is 5
The occurrence count of the largest number is 4
```

***5.42** (*Financial application: find the sales amount*) Rewrite Programming Exercise 5.39 as follows:

- Use a `for` loop instead of a `do-while` loop.
- Let the user enter `COMMISSION_SOUGHT` instead of fixing it as a constant.

***5.43** (*Math: combinations*) Write a program that displays all possible combinations for picking two numbers from integers `1` to `7`. Also display the total number of all combinations.

```
1 2
1 3
...
...

The total number of all combinations is 21
```

***5.44** (*Computer architecture: bit-level operations*) A `short` value is stored in `16` bits. Write a program that prompts the user to enter a short integer and displays the `16` bits for the integer. Here are sample runs:

```
Enter an integer: 5 ↵Enter
The bits are 0000000000000101
```

```
Enter an integer: -5 ↵Enter
The bits are 1111111111111011
```

(*Hint:* You need to use the bitwise right shift operator (`>>`) and the bitwise `AND` operator (`&`), which are covered in Appendix G, Bitwise Operations.)

****5.45** (*Statistics: compute mean and standard deviation*) In business applications, you are often asked to compute the mean and standard deviation of data. The mean is simply the average of the numbers. The standard deviation is a statistic that tells you how tightly all the various data are clustered around the mean in a set of data. For example, what is the average age of the students in a class? How close are the ages? If all the students are the same age, the deviation is 0.

Write a program that prompts the user to enter 10 numbers and displays the mean and standard deviations of these numbers using the following formula:

$$\text{mean} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1 + x_2 + \cdots + x_n}{n} \qquad \text{deviation} = \sqrt{\frac{\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n - 1}}$$

Here is a sample run:

```
Enter 10 numbers: 1 2 3 4.5 5.6 6 7 8 9 10 ↵Enter
The mean is 5.61
The standard deviation is 2.99794
```

***5.46** (*Reverse a string*) Write a program that prompts the user to enter a string and displays the string in reverse order.

```
Enter a string: ABCD ↵Enter
The reversed string is DCBA
```

***5.47** (*Business: check ISBN-13*) **`ISBN-13`** is a new standard for identifying books. It uses 13 digits $d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}d_{11}d_{12}d_{13}$. The last digit d_{13} is a checksum, which is calculated from the other digits using the following formula:

$$10 - (d_1 + 3d_2 + d_3 + 3d_4 + d_5 + 3d_6 + d_7 + 3d_8 + d_9 + 3d_{10} + d_{11} + 3d_{12})\%10$$

If the checksum is **`10`**, replace it with **`0`**. Your program should read the input as a string. Display “invalid input” if the input is invalid. Here are sample runs:

```
Enter the first 12 digits of an ISBN-13 as a string: 978013213080 ↵Enter
The ISBN-13 number is 9780132130806
```

```
Enter the first 12 digits of an ISBN-13 as a string: 978013213079 ↵Enter
The ISBN-13 number is 9780132130790
```

```
Enter the first 12 digits of an ISBN-13 as a string: 97801320 ↵Enter
97801320 is an invalid input
```

***5.48** (*Process string*) Write a program that prompts the user to enter a string and displays the characters at odd positions. Here is a sample run:

```
Enter a string: Beijing Chicago ↵Enter
BiigCiao
```

***5.49** (*Count vowels and consonants*) Assume that the letters **`A`**, **`E`**, **`I`**, **`O`**, and **`U`** are vowels. Write a program that prompts the user to enter a string, and displays the number of vowels and consonants in the string.

```
Enter a string: Programming is fun ↵Enter
The number of vowels is 5
The number of consonants is 11
```

***5.50** (*Count uppercase letters*) Write a program that prompts the user to enter a string and displays the number of the uppercase letters in the string.

```
Enter a string: Welcome to Java ↵Enter
The number of uppercase letters is 2
```

*5.51 (_Longest common prefix_) Write a program that prompts the user to enter two strings and displays the largest common prefix of the two strings. Here are some sample runs:

```
Enter the first string: Welcome to C++ ↵Enter
Enter the second string: Welcome to programming ↵Enter
The common prefix is Welcome to
```

```
Enter the first string: Atlanta ↵Enter
Enter the second string: Macon ↵Enter
Atlanta and Macon have no common prefix
```

CHAPTER

6

METHODS

Objectives

- To define methods with formal parameters (§6.2).
- To invoke methods with actual parameters (i.e., arguments) (§6.2).
- To define methods with a return value (§6.3).
- To define methods without a return value and distinguish the differences between void methods and value-returning methods (§6.4).
- To pass arguments by value (§6.5).
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain (§6.6).
- To write a method that converts hexadecimals to decimals (§6.7).
- To use method overloading and understand ambiguous overloading (§6.8).
- To determine the scope of variables (§6.9).
- To apply the concept of method abstraction in software development (§6.10).
- To design and implement methods using stepwise refinement (§6.11).

6.1 Introduction

Key Point

Methods can be used to define reusable code and organize and simplify coding, and make code easy to maintain.

problem

Suppose you need to find the sum of integers from **1** to **10**, **20** to **37**, and **35** to **49**, respectively. You may write the code as follows:

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 37; i++)
  sum += i;
System.out.println("Sum from 20 to 37 is " + sum);

sum = 0;
for (int i = 35; i <= 49; i++)
  sum += i;
System.out.println("Sum from 35 to 49 is " + sum);
```

why methods?

You may have observed that computing these sums from **1** to **10**, **20** to **37**, and **35** to **49** are very similar, except that the starting and ending integers are different. Wouldn't it be nice if we could write the common code once and reuse it? We can do so by defining a method and invoking it.

The preceding code can be simplified as follows:

Listing `MethodDemo.java`

define `sum` method

main method

invoke `sum`

```
 1  public static int sum(int i1, int i2) {
 2    int result = 0;
 3    for (int i = i1; i <= i2; i++)
 4      result += i;
 5
 6    return result;
 7  }
 8
 9  public static void main(String[] args) {
10    System.out.println("Sum from 1 to 10 is " + sum(1, 10));
11    System.out.println("Sum from 20 to 37 is " + sum(20, 37));
12    System.out.println("Sum from 35 to 49 is " + sum(35, 49));
13  }
```

Lines 1–7 define the method named **sum** with two parameters **i1** and **i2**. The statements in the **main** method invoke **sum(1, 10)** to compute the sum from **1** to **10**, **sum(20, 37)** to compute the sum from **20** to **37**, and **sum(35, 49)** to compute the sum from **35** to **49**.

method

A *method* is a collection of statements grouped together to perform an operation. In earlier chapters you have used predefined methods such as **System.out.println**, **System.exit**, **Math.pow**, and **Math.random**. These methods are defined in the Java library. In this chapter, you will learn how to define your own methods and apply method abstraction to solve complex problems.

Check Point

6.1.1 What are the benefits of using a method?

6.2 Defining a Method

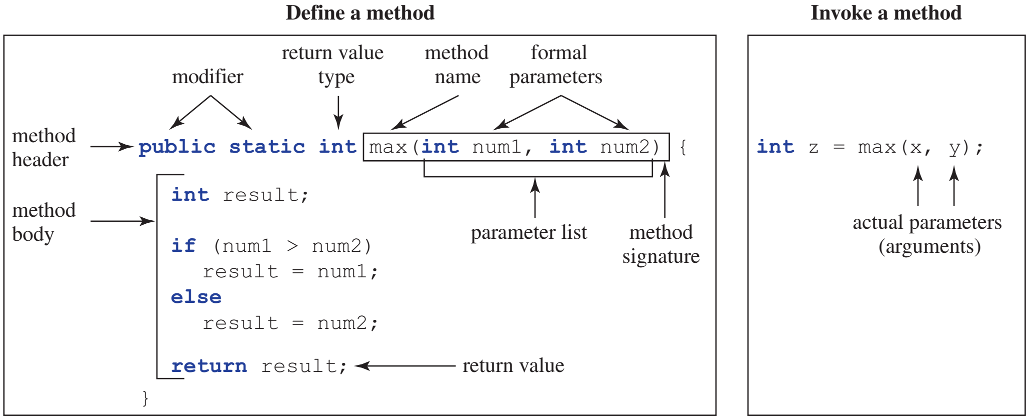
Key Point

A method definition consists of method name, parameters, return value type, and body.

The syntax for defining a method is as follows:

```
modifier returnValueType methodName(list of parameters) {
  // Method body;
}
```

Let's look at a method defined to find the larger between two integers. This method, named `max`, has two `int` parameters, `num1` and `num2`, the larger of which is returned by the method. Figure 6.1 illustrates the components of this method.



```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

```
int z = max(x, y);
```

FIGURE 6.1 A method definition consists of a method header and a method body.

method header
modifier

The *method header* specifies the *modifiers*, *return value type*, *method name*, and *parameters* of the method. The `static` modifier is used for all the methods in this chapter. The reason for using it will be discussed in Chapter 9, Objects and Classes.

A method may return a value. The `returnValueType` is the data type of the value the method returns. Some methods perform desired operations without returning a value. In this case, the `returnValueType` is the keyword `void`. For example, the `returnValueType` is `void` in the `main` method, as well as in `System.exit`, and `System.out.println`. If a method returns a value, it is called a *value-returning method;* otherwise, it is called a *void method.*

value-returning method
void method

formal parameter
parameter
actual parameter
argument
parameter list
method signature

The variables defined in the method header are known as *formal parameters* or simply *parameters*. A parameter is like a placeholder: when a method is invoked, you pass a value to the parameter. This value is referred to as an *actual parameter or argument*. The *parameter list* refers to the method's type, order, and the number of parameters. The method name and the parameter list together constitute the *method signature*. Parameters are optional; that is, a method may contain no parameters. For example, the `Math.random()` method has no parameters.

The method body contains a collection of statements that implement the method. The method body of the `max` method uses an `if` statement to determine which number is larger and return the value of that number. In order for a value-returning method to return a result, a return statement using the keyword `return` is *required.* The method terminates when a return statement is executed.

Note

Some programming languages refer to methods as *procedures* and *functions.* In those languages, a value-returning method is called a *function* and a void method is called a *procedure.*

Caution

In the method header, you need to declare each parameter separately. For instance, `max(int num1, int num2)` is correct, but `max(int num1, num2)` is wrong.

Note

define vs. declare

We say "*define* a method" and "*declare* a variable." We are making a subtle distinction here. A definition defines what the defined item is, but a declaration usually involves allocating memory to store data for the declared item.

Check Point

6.2.1 How do you simplify the `max` method in Listing 6.1 using the conditional operator?

6.2.2 Define the terms parameter, argument, and method signature.

6.3 Calling a Method

Key Point

Calling a method executes the code in the method.

caller

In a method definition, you define what the method is to do. To execute the method, you have to *call* or *invoke* it. The program that calls the function is called a *caller*. There are two ways to call a method, depending on whether the method returns a value or not.

If a method returns a value, a call to the method is usually treated as a value. For example,

```
int larger = max(3, 4);
```

calls `max(3, 4)` and assigns the result of the method to the variable `larger`. Another example of a call that is treated as a value is

```
System.out.println(max(3, 4));
```

which prints the return value of the method call `max(3, 4)`.

If a method returns `void`, a call to the method must be a statement. For example, the method `println` returns `void`. The following call is a statement:

```
System.out.println("Welcome to Java!");
```

Note

A value-returning method can also be invoked as a statement in Java. In this case, the caller simply ignores the return value. This is not often done, but it is permissible if the caller is not interested in the return value.

When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

Listing 6.1 presents a complete program that is used to test the `max` method.

VideoNote
Define/invoke `max` method

Listing 6.1 TestMax.java

main method

invoke max

define method

```
public class TestMax {
  /** Main method */
  public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j);
    System.out.println("The maximum of " + i +
      " and " + j + " is " + k);
  }

  /** Return the max of two numbers */
  public static int max(int num1, int num2) {
    int result;

    if (num1 > num2)
      result = num1;
    else
      result = num2;

    return result;
  }
}
```

```
The maximum of 5 and 2 is 5
```

	line#	i	j	k	num1	num2	result
	4	5					
	5		2				
Invoking max	12				5	2	
	13						undefined
	16						5
	6			5			

This program contains the `main` method and the `max` method. The `main` method is just like any other method, except that it is invoked by the JVM to start the program.

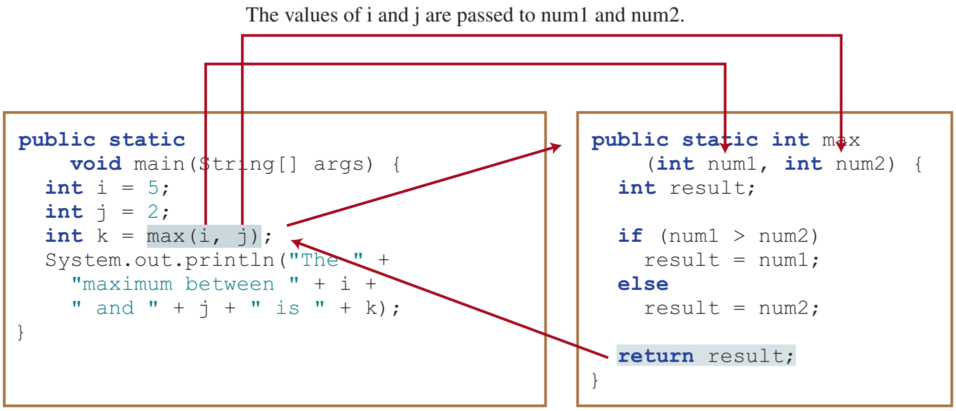
`main` method

The `main` method's header is always the same. Like the one in this example, it includes the modifiers `public` and `static`, return value type `void`, method name `main`, and a parameter of the `String[]` type. `String[]` indicates the parameter is an array of `String`, a subject addressed in Chapter 7.

The statements in `main` may invoke other methods that are defined in the class that contains the `main` method or in other classes. In this example, the `main` method invokes `max(i, j)`, which is defined in the same class with the `main` method.

`max` method

When the `max` method is invoked (line 6), variable `i`'s value `5` is passed to `num1` and variable `j`'s value `2` is passed to `num2` in the `max` method. The flow of control transfers to the `max` method and the `max` method is executed. When the `return` statement in the `max` method is executed, the `max` method returns the control to its caller (in this case, the caller is the `main` method). This process is illustrated in Figure 6.2.



The values of i and j are passed to num1 and num2.

```
public static
   void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);
  System.out.println("The " +
    "maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max
    (int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

FIGURE 6.2 When the `max` method is invoked, the flow of control transfers to it. Once the `max` method is finished, it returns control back to the caller.

Caution

A `return` statement is required for a value-returning method. The method given in (a) is logically correct, but it has a compile error because the Java compiler thinks this method might not return a value.

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return –1;
}
```

(a)

Should be

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return –1;
}
```

(b)

To fix this problem, delete `if (n < 0)` in (a), so the compiler will see a `return` statement to be reached regardless of how the `if` statement is evaluated, as shown in (b).

reusing method

Note

Methods enable code sharing and reuse. The `max` method can be invoked from any class, not just `TestMax`. If you create a new class, you can invoke the `max` method using `ClassName.methodName` (i.e., `TestMax.max`).

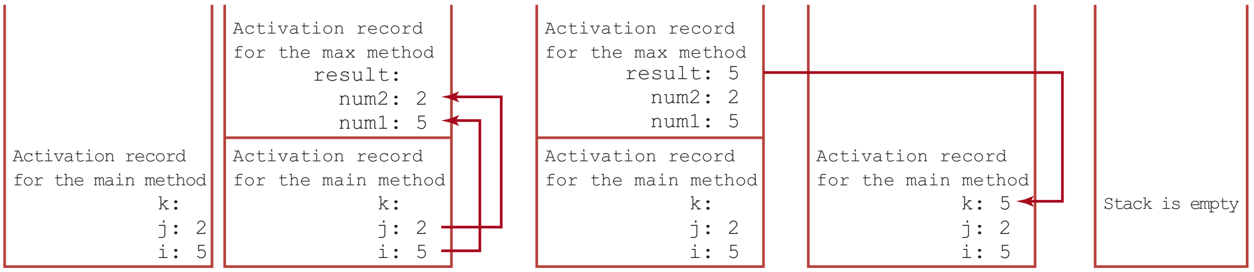
activation record

call stack

Each time a method is invoked, the system creates an *activation record* that stores parameters and variables for the method and places the activation record in an area of memory known as a *call stack*. A call stack is also known as an *execution stack*, *runtime stack*, or *machine stack* and it is often shortened to just "the stack." When a method calls another method, the caller's activation record is kept intact and a new activation record is created for the new method called. When a method finishes its work and returns to its caller, its activation record is removed from the call stack.

A call stack stores the activation records in a last-in, first-out fashion: The activation record for the method that is invoked last is removed first from the stack. For example, suppose method `m1` calls method `m2`, and `m2` calls method `m3`. The runtime system pushes `m1`'s activation record into the stack, then `m2`'s, and then `m3`'s. After `m3` is finished, its activation record is removed from the stack. After `m2` is finished, its activation record is removed from the stack. After `m1` is finished, its activation record is removed from the stack.

Understanding call stacks helps you to comprehend how methods are invoked. The variables defined in the `main` method in Listing 6.1 are `i`, `j`, and `k`. The variables defined in the `max` method are `num1`, `num2`, and `result`. The variables `num1` and `num2` are defined in the method signature and are parameters of the `max` method. Their values are passed through method invocation. Figure 6.3 illustrates the activation records for method calls in the stack.


Activation record
for the main method
k:
j: 2
i: 5
Activation record
for the max method
result:
num2: 2
num1: 5
Activation record
for the main method
k:
j: 2
i: 5
Activation record
for the max method
result: 5
num2: 2
num1: 5
Activation record
for the main method
k:
j: 2
i: 5
Activation record
for the main method
k: 5
j: 2
i: 5
Stack is empty

(1) The `main` method is invoked.

(2) The `max` method is invoked.

(3) The `max` method is being executed.

(4) The `max` method is finished and the return value is sent to `k`.

(5) The `main` method is finished.

Figure 6.3 When the `max` method is invoked, the flow of control transfers to the `max` method. Once the `max` method is finished, it returns control back to the caller.

Check Point

6.3.1 How do you define a method? How do you invoke a method?

6.3.2 Reformat the following program according to the programming style and documentation guidelines proposed in Section 1.9, Programming Style and Documentation. Use the end-of-line brace style.

```
public class Test {
  public static double method(double i, double j)
  {
    while (i < j) {
      j--;
    }
    return j;
  }
}
```

6.4 `void` vs. Value-Returning Methods

A **`void`** *method does not return a value.*

Key Point

The preceding section gives an example of a value-returning method. This section shows how to define and invoke a **`void`** method. Listing 6.2 gives a program that defines a method named **`printGrade`** and invokes it to print the grade for a given score.

VideoNote

Use `void` method

LISTING 6.2 `TestVoidMethod.java`

```
 1  public class TestVoidMethod {
 2    public static void main(String[] args) {
 3      System.out.print("The grade is ");
 4      printGrade(78.5);
 5
 6      System.out.print("The grade is ");
 7      printGrade(59.5);
 8    }
 9
10    public static void printGrade(double score) {
11      if (score >= 90.0) {
12        System.out.println('A');
13      }
14      else if (score >= 80.0) {
15        System.out.println('B');
16      }
17      else if (score >= 70.0) {
18        System.out.println('C');
19      }
20      else if (score >= 60.0) {
21        System.out.println('D');
22      }
23      else {
24        System.out.println('F');
25      }
26    }
27  }
```

`main` method

invoke `printGrade`

`printGrade` method

```
The grade is C
The grade is F
```

invoke `void` method

The **`printGrade`** method is a **`void`** method because it does not return any value. A call to a **`void`** method must be a statement. Therefore, it is invoked as a statement in line 4 in the **`main`** method. Like any Java statement, it is terminated with a semicolon.

void vs. value-returned

To see the differences between a void and value-returning method, let's redesign the **`printGrade`** method to return a value. The new method, which we call **`getGrade`**, returns the grade as given in Listing 6.3.

Listing 6.3 TestReturnGradeMethod.java

main method

invoke getGrade

getGrade method

```
 1  public class TestReturnGradeMethod {
 2    public static void main(String[] args) {
 3      System.out.print("The grade is " + getGrade(78.5));
 4      System.out.print("\nThe grade is " + getGrade(59.5));
 5    }
 6
 7    public static char getGrade(double score) {
 8      if (score >= 90.0)
 9        return 'A';
10      else if (score >= 80.0)
11        return 'B';
12      else if (score >= 70.0)
13        return 'C';
14      else if (score >= 60.0)
15        return 'D';
16      else
17        return 'F';
18    }
19  }
```

```
The grade is C
The grade is F
```

The **getGrade** method defined in lines 7–18 returns a character grade based on the numeric score value. The caller invokes this method in lines 3 and 4.

The **getGrade** method can be invoked by a caller wherever a character may appear. The **printGrade** method does not return any value, so it must be invoked as a statement.

return in void method

Note

A **return** statement is not needed for a **void** method, but it can be used for terminating the method and returning to the method's caller. The syntax is simply

```
return;
```

This is not often done, but sometimes it is useful for circumventing the normal flow of control in a **void** method. For example, the following code has a return statement to terminate the method when the score is invalid:

```
public static void printGrade(double score) {
  if (score < 0 || score > 100) {
    System.out.println("Invalid score");
    return;
  }

  if (score >= 90.0) {
    System.out.println('A');
  }
  else if (score >= 80.0) {
    System.out.println('B');
  }
  else if (score >= 70.0) {
    System.out.println('C');
  }
  else if (score >= 60.0) {
    System.out.println('D');
  }
  else {
    System.out.println('F');
  }
}
```

6.4.1 True or false? A call to a method with a `void` return type is always a statement itself, but a call to a value-returning method cannot be a statement by itself.

6.4.2 What is the `return` type of a `main` method?

6.4.3 What would be wrong with not writing a `return` statement in a value-returning method? Can you have a `return` statement in a `void` method? Does the `return` statement in the following method cause syntax errors?

```
public static void xMethod(double x, double y) {
  System.out.println(x + y);
  return x + y;
}
```

6.4.4 Write method headers (not the bodies) for the following methods:

a. Return a sales commission, given the sales amount and the commission rate.

b. Display the calendar for a month, given the month and year.

c. Return a square root of a number.

d. Test whether a number is even, and returning `true` if it is.

e. Display a message a specified number of times.

f. Return the monthly payment, given the loan amount, number of years, and annual interest rate.

g. Return the corresponding uppercase letter, given a lowercase letter.

6.4.5 Identify and correct the errors in the following program:

```
public class Test {
  public static method1(int n, m) {
    n += m;
    method2(3.4);
  }

  public static int method2(int n) {
    if (n > 0) return 1;
    else if (n == 0) return 0;
    else if (n < 0) return -1;
  }
}
```

6.5 Passing Arguments by Values

The arguments are passed by value to parameters when invoking a method.

Key Point

parameter order association

The power of a method is its ability to work with parameters. You can use `println` to print any string, and `max` to find the maximum of any two `int` values. When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method signature. This is known as *parameter order association*. For example, the following method prints a message `n` times:

```
public static void nPrintln(String message, int n) {
  for (int i = 0; i < n; i++)
    System.out.println(message);
}
```

You can use `nPrintln("Hello", 3)` to print `Hello` three times. The `nPrintln("Hello", 3)` statement passes the actual string parameter `Hello` to the parameter `message`, passes `3` to `n`, and prints `Hello` three times. However, the statement `nPrintln(3, "Hello")` would be wrong. The data type of `3` does not match the data type for the first parameter, `message`, nor does the second argument, `Hello`, match the second parameter, `n`.

> **Caution**
>
> The arguments must match the parameters in *order*, *number*, and *compatible type*, as defined in the method signature. Compatible type means you can pass an argument to a parameter without explicit casting, such as passing an `int` value argument to a `double` value parameter.

pass-by-value

When you invoke a method with an argument, the value of the argument is passed to the parameter. This is referred to as *pass-by-value*. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method. As given in Listing 6.4, the value of `x` (`1`) is passed to the parameter `n` to invoke the `increment` method (line 5). The parameter `n` is incremented by `1` in the method (line 10), but `x` is not changed no matter what the method does.

Listing 6.4 Increment.java

invoke increment

increment n

```
 1  public class Increment {
 2    public static void main(String[] args) {
 3      int x = 1;
 4      System.out.println("Before the call, x is " + x);
 5      increment(x);
 6      System.out.println("After the call, x is " + x);
 7    }
 8
 9    public static void increment(int n) {
10      n++;
11      System.out.println("n inside the method is " + n);
12    }
13  }
```

```
Before the call, x is 1
n inside the method is 2
After the call, x is 1
```

Listing 6.5 gives another program that demonstrates the effect of passing by value. The program creates a method for swapping two variables. The `swap` method is invoked by passing two arguments. Interestingly, the values of the arguments are not changed after the method is invoked.

Listing 6.5 TestPassByValue.java

```
public class TestPassByValue {
  /** Main method */
  public static void main(String[] args) {
    // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;

    System.out.println("Before invoking the swap method, num1 is " +
      num1 + " and num2 is " + num2);
10
11      // Invoke the swap method to attempt to swap two variables
12      swap(num1, num2);
13
14      System.out.println("After invoking the swap method, num1 is " +
15        num1 + " and num2 is " + num2);
16    }
17
18    /** Swap two variables */
19    public static void swap(int n1, int n2) {
20      System.out.println("\tInside the swap method");
21      System.out.println("\t\tBefore swapping, n1 is " + n1
22        + " and n2 is " + n2);
23
24      // Swap n1 with n2
25      int temp = n1;
26      n1 = n2;
27      n2 = temp;
28
29      System.out.println("\t\tAfter swapping, n1 is " + n1
30        + " and n2 is " + n2);
31    }
32  }
```

false swap

```
Before invoking the swap method, num1 is 1 and num2 is 2
  Inside the swap method
    Before swapping, n1 is 1 and n2 is 2
    After swapping, n1 is 2 and n2 is 1
After invoking the swap method, num1 is 1 and num2 is 2
```

Before the **swap** method is invoked (line 12), **num1** is **1** and **num2** is **2**. After the **swap** method is invoked, **num1** is still **1** and **num2** is still **2**. Their values have not been swapped. As shown in Figure 6.4, the values of the arguments **num1** and **num2** are passed to **n1** and **n2**, but **n1** and **n2** have their own memory locations independent of **num1** and **num2**. Therefore, changes in **n1** and **n2** do not affect the contents of **num1** and **num2**.

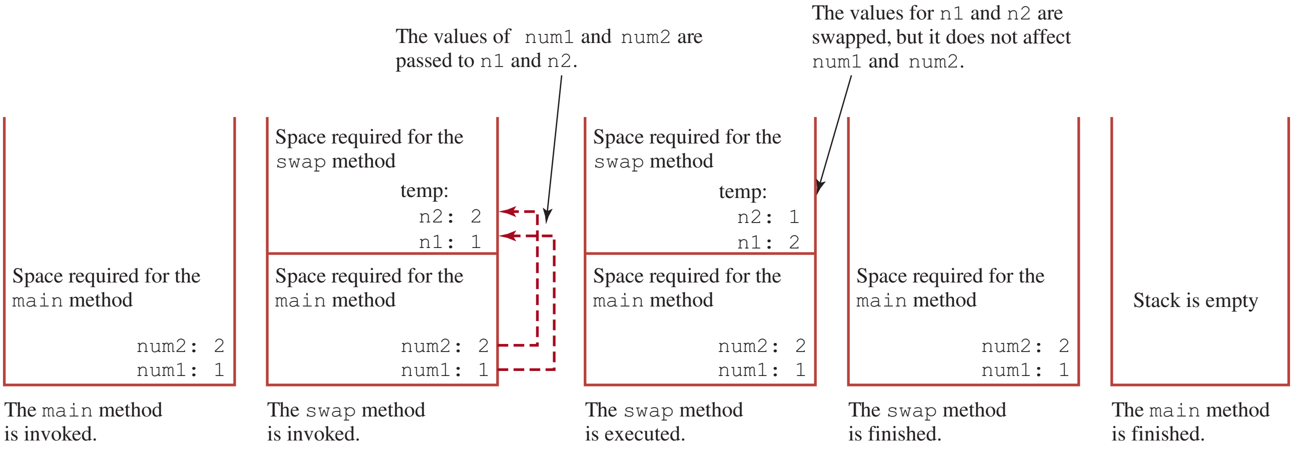

The values of num1 and num2 are passed to n1 and n2.
The values for n1 and n2 are swapped, but it does not affect num1 and num2.
Space required for the main method
num2: 2
num1: 1
Space required for the swap method
temp:
n2: 2
n1: 1
Space required for the main method
num2: 2
num1: 1
Space required for the swap method
temp:
n2: 1
n1: 2
Space required for the main method
num2: 2
num1: 1
Space required for the main method
num2: 2
num1: 1
Stack is empty
The main method is invoked.
The swap method is invoked.
The swap method is executed.
The swap method is finished.
The main method is finished.

FIGURE 6.4 The values of the variables are passed to the method's parameters.

Another twist is to change the parameter name `n1` in `swap` to `num1`. What effect does this have? No change occurs, because it makes no difference whether the parameter and the argument have the same name. The parameter is a variable in the method with its own memory space. The variable is allocated when the method is invoked, and it disappears when the method is returned to its caller.

Note

For simplicity, Java programmers often say *passing* `x` *to* `y`, which actually means *passing the value of argument* `x` *to parameter* `y`.

Check Point

6.5.1 How is an argument passed to a method? Can the argument have the same name as its parameter?

6.5.2 Identify and correct the errors in the following program:

```java
public class Test {
  public static void main(String[] args) {
    nPrintln(5, "Welcome to Java!");
  }

  public static void nPrintln(String message, int n) {
    int n = 1;
    for (int i = 0; i < n; i++)
      System.out.println(message);
  }
}
```

6.5.3 What is pass-by-value? Show the result of the following programs.

6.5.4 For (a) in the preceding question, show the contents of the activation records in the call stack just before the method `max` is invoked, just as `max` is entered, just before `max` is returned, and right after `max` is returned.

```java
public class Test {
  public static void main(String[] args) {
    int max = 0;
    max(1, 2, max);
    System.out.println(max);
  }

  public static void max(
      int value1, int value2, int max) {
    if (value1 > value2)
      max = value1;
    else
      max = value2;
  }
}
```

(a)

```java
public class Test {
  public static void main(String[] args) {
    int i = 1;
    while (i <= 6) {
      method1(i, 2);
      i++;
    }
  }

  public static void method1(
      int i, int num) {
    for (int j = 1; j <= i; j++) {
      System.out.print(num + " ");
      num *= 2;
    }

    System.out.println();
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
    // Initialize times
    int times = 3;
    System.out.println("Before the call,"
      + " variable times is " + times);

    // Invoke nPrintln and display times
    nPrintln("Welcome to Java!", times);
    System.out.println("After the call,"
      + " variable times is " + times);
  }

  // Print the message n times
  public static void nPrintln(
      String message, int n) {
    while (n > 0) {
      System.out.println("n = " + n);
      System.out.println(message);
      n--;
    }
  }
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    int i = 0;
    while (i <= 4) {
      method1(i);
      i++;
    }

    System.out.println("i is " + i);
  }

  public static void method1(int i) {
    do {
      if (i % 3 != 0)
        System.out.print(i + " ");
      i--;
    }
    while (i >= 1);

    System.out.println();
  }
}
```

(d)

6.6 Modularizing Code

Modularizing makes the code easy to maintain and debug and enables the code to be reused.

Key Point

Methods can be used to reduce redundant code and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.

Listing 5.9 gives a program that prompts the user to enter two integers and displays their greatest common divisor. You can rewrite the program using a method, as given in Listing 6.6.

VideoNote
Modularize code

Listing 6.6 GreatestCommonDivisorMethod.java

```
import java.util.Scanner;

public class GreatestCommonDivisorMethod {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter first integer: ");
    int n1 = input.nextInt();
    System.out.print("Enter second integer: ");
    int n2 = input.nextInt();

    System.out.println("The greatest common divisor for " + n1 +
      " and " + n2 + " is " + gcd(n1, n2));
  }
```

invoke gcd

compute gcd

return gcd

```

  /** Return the gcd of two integers */
  public static int gcd(int n1, int n2) {
    int gcd = 1; // Initial gcd is 1
    int k = 2; // Possible gcd

    while (k <= n1 && k <= n2) {
      if (n1 % k == 0 && n2 % k == 0)
        gcd = k; // Update gcd
      k++;
    }

    return gcd; // Return gcd
  }
}
```

```
Enter first integer: 45 ↵Enter
Enter second integer: 75 ↵Enter
The greatest common divisor for 45 and 75 is 15
```

By encapsulating the code for obtaining the gcd in a method, this program has several advantages:

1. It isolates the problem for computing the gcd from the rest of the code in the main method. Thus, the logic becomes clear, and the program is easier to read.
2. The errors on computing the gcd are confined in the **gcd** method, which narrows the scope of debugging.
3. The **gcd** method now can be reused by other programs.

Listing 6.7 applies the concept of code modularization to improve Listing 5.15, PrimeNumber.java.

LISTING 6.7 PrimeNumberMethod.java

invoke printPrimeNumbers

printPrimeNumbers method

invoke isPrime

```
public class PrimeNumberMethod {
  public static void main(String[] args) {
    System.out.println("The first 50 prime numbers are \n");
    printPrimeNumbers(50);
  }

  public static void printPrimeNumbers(int numberOfPrimes) {
    final int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
    int count = 0; // Count the number of prime numbers
    int number = 2; // A number to be tested for primeness

    // Repeatedly find prime numbers
    while (count < numberOfPrimes) {
      // Print the prime number and increase the count
      if (isPrime(number)) {
        count++; // Increase the count

        if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
          // Print the number and advance to the new line
          System.out.printf("%-5d\n", number);
        }
        else
          System.out.printf("%-5d", number);
      }

      // Check whether the next number is prime
      number++;
    }
  }

  /** Check whether number is prime */
  public static boolean isPrime(int number) {
    for (int divisor = 2; divisor <= number / 2; divisor++) {
      if (number % divisor == 0) { // If true, number is not prime
        return false; // Number is not a prime
      }
    }

    return true; // Number is prime
  }
}
```

`isPrime` method

```
The first 50 prime numbers are

2    3    5    7    11   13   17   19   23   29
31   37   41   43   47   53   59   61   67   71
73   79   83   89   97   101  103  107  109  113
127  131  137  139  149  151  157  163  167  173
179  181  191  193  197  199  211  223  227  229
```

We divided a large problem into two subproblems: determining whether a number is a prime, and printing the prime numbers. As a result, the new program is easier to read and easier to debug. Moreover, the methods `printPrimeNumbers` and `isPrime` can be reused by other programs.

Check Point

6.6.1 Trace the `gcd` method to find the return value for `gcd(4, 6)`.

6.6.2 Trace the `isPrime` method to find the return value for `isPrime(25)`.

6.7 Case Study: Converting Hexadecimals to Decimals

Key Point

This section presents a program that converts a hexadecimal number into a decimal number.

Listing 5.11, Dec2Hex.java, gives a program that converts a decimal to a hexadecimal. How would you convert a hex number into a decimal?

Given a hexadecimal number $h_n h_{n-1} h_{n-2} \ldots h_2 h_1 h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots$$
$$+ h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

For example, the hex number `AB8C` is

$$10 \times 16^3 + 11 \times 16^2 + 8 \times 16^1 + 12 \times 16^0 = 43916$$

Our program will prompt the user to enter a hex number as a string and convert it into a decimal using the following method:

```
public static int hexToDecimal(String hex)
```

A brute-force approach is to convert each hex character into a decimal number, multiply it by 16^i for a hex digit at the **`i`**'s position, and then add all the items together to obtain the equivalent decimal value for the hex number.

Note that

$$\begin{aligned} &h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \cdots + h_1 \times 16^1 + h_0 \times 16^0 \\ &\quad = (\ldots((h_n \times 16 + h_{n-1}) \times 16 + h_{n-2}) \times 16 + \cdots + h_1) \times 16 + h_0 \end{aligned}$$

This observation, known as the Horner's algorithm, leads to the following efficient code for converting a hex string to a decimal number:

```java
int decimalValue = 0;
for (int i = 0; i < hex.length(); i++) {
  char hexChar = hex.charAt(i);
  decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
}
```

Here is a trace of the algorithm for hex number **`AB8C`**:

	`i`	`hexChar`	`hexCharToDecimal (hexChar)`	`decimalValue`
Before the loop				`0`
After the 1st iteration	`0`	`A`	`10`	`10`
After the 2nd iteration	`1`	`B`	`11`	`10 * 16 + 11`
After the 3rd iteration	`2`	`8`	`8`	`(10 * 16 + 11) * 16 + 8`
After the 4th iteration	`3`	`C`	`12`	`((10 * 16 + 11) * 16 + 8) * 16 + 12`

Listing 6.8 gives the complete program.

LISTING 6.8 Hex2Dec.java

input string

hex to decimal

```java
import java.util.Scanner;

public class Hex2Dec {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a string
    System.out.print("Enter a hex number: ");
    String hex = input.nextLine();

    System.out.println("The decimal value for hex number "
      + hex + " is " + hexToDecimal(hex.toUpperCase()));
  }

  public static int hexToDecimal(String hex) {
    int decimalValue = 0;
    for (int i = 0; i < hex.length(); i++) {
      char hexChar = hex.charAt(i);
      decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
22        }
23
24        return decimalValue;
25      }
26
27      public static int hexCharToDecimal(char ch) {
28        if (ch >= 'A' && ch <= 'F')
29          return 10 + ch - 'A';
30        else // ch is '0', '1', ..., or '9'
31          return ch - '0';
32      }
33    }
```

hex char to decimal
check uppercase

```
Enter a hex number: AB8C ↵Enter
The decimal value for hex number AB8C is 43916
```

```
Enter a hex number: af71 ↵Enter
The decimal value for hex number af71 is 44913
```

The program reads a string from the console (line 11) and invokes the `hexToDecimal` method to convert a hex string to decimal number (line 14). The characters can be in either lowercase or uppercase. They are converted to uppercase before invoking the `hexToDecimal` method.

The `hexToDecimal` method is defined in lines 17–25 to return an integer. The length of the string is determined by invoking `hex.length()` in line 19.

The `hexCharToDecimal` method is defined in lines 27–32 to return a decimal value for a hex character. The character can be in either lowercase or uppercase. Recall that to subtract two characters is to subtract their Unicodes. For example, `'5' – '0'` is `5`.

Check Point

6.7.1 What is `hexCharToDecimal('B'))`?
What is `hexCharToDecimal('7'))`?
What is `hexToDecimal("A9"))`?

6.8 Overloading Methods

Key Point

Overloading methods enable you to define the methods with the same name as long as their parameter lists are different.

The `max` method used earlier works only with the `int` data type. But what if you need to determine which of the two floating-point numbers has the maximum value? The solution is to create another method with the same name but different parameters, as shown in the following code:

```
public static double max(double num1, double num2) {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

method overloading

If you call `max` with `int` parameters, the `max` method that expects `int` parameters will be invoked; and if you call `max` with `double` parameters, the `max` method that expects `double` parameters will be invoked. This is referred to as *method overloading*; that is, two methods have the same name but different parameter lists within one class. The Java compiler determines which method to use based on the method signature.

Listing 6.9 is a program that creates three methods. The first finds the maximum integer, the second finds the maximum double, and the third finds the maximum among three double values. All three methods are named `max`.

Listing 6.9 `TestMethodOverloading.java`

```
 1  public class TestMethodOverloading {
 2    /** Main method */
 3    public static void main(String[] args) {
 4      // Invoke the max method with int parameters
 5      System.out.println("The maximum of 3 and 4 is "
 6        + max(3, 4));
 7
 8      // Invoke the max method with the double parameters
 9      System.out.println("The maximum of 3.0 and 5.4 is "
10        + max(3.0, 5.4));
11
12      // Invoke the max method with three double parameters
13      System.out.println("The maximum of 3.0, 5.4, and 10.14 is "
14        + max(3.0, 5.4, 10.14));
15    }
16
17    /** Return the max of two int values */
18    public static int max(int num1, int num2) {
19      if (num1 > num2)
20        return num1;
21      else
22        return num2;
23    }
24
25    /** Find the max of two double values */
26    public static double max(double num1, double num2) {
27      if (num1 > num2)
28        return num1;
29      else
30        return num2;
31    }
32
33    /** Return the max of three double values */
34    public static double max(double num1, double num2, double num3) {
35      return max(max(num1, num2), num3);
36    }
37  }
```

overloaded `max`

overloaded `max`

overloaded `max`

```
The maximum of 3 and 4 is 4
The maximum of 3.0 and 5.4 is 5.4
The maximum of 3.0, 5.4, and 10.14 is 10.14
```

When calling `max(3, 4)` (line 6), the `max` method for finding the maximum of two integers is invoked. When calling `max(3.0, 5.4)` (line 10), the `max` method for finding the maximum of two doubles is invoked. When calling `max(3.0, 5.4, 10.14)` (line 14), the `max` method for finding the maximum of three double values is invoked.

Can you invoke the `max` method with an `int` value and a `double` value, such as `max(2, 2.5)`? If so, which of the `max` methods is invoked? The answer to the first question is yes. The answer to the second question is that the `max` method for finding the maximum of two `double` values is invoked. The argument value `2` is automatically converted into a `double` value and passed to this method.

You may be wondering why the method `max(double, double)` is not invoked for the call `max(3, 4)`. Both `max(double, double)` and `max(int, int)` are possible matches for `max(3, 4)`. The Java compiler finds the method that best matches a method invocation. Since the method `max(int, int)` is a better match for `max(3, 4)` than `max(double, double)`, `max(int, int)` is used to invoke `max(3, 4)`.

Tip

Overloading methods can make programs clearer and more readable. Methods that perform the same function with different types of parameters should be given the same name.

Note

Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers or return types.

Note

ambiguous invocation

Sometimes there are two or more possible matches for the invocation of a method, but the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*. Ambiguous invocation causes a compile error. Consider the following code:

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
  }

  public static double max(int num1, double num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }

  public static double max(double num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
}
```

Both `max(int, double)` and `max(double, int)` are possible candidates to match `max(1, 2)`. Because neither is more specific than the other, the invocation is ambiguous, resulting in a compile error.

Check Point

6.8.1 What is method overloading? Is it permissible to define two methods that have the same name but different parameter types? Is it permissible to define two methods in a class that have identical method names and parameter lists, but different return value types or different modifiers?

6.8.2 What is wrong in the following program?

```
public class Test {
  public static void method(int x) {
  }

  public static int method(int y) {
      return y;
    }
  }
```

6.8.3 Given two method definitions,

```
public static double m(double x, double y)

public static double m(int x, double y)
```

tell which of the two methods is invoked for:

a. `double z = m(4, 5);`

b. `double z = m(4, 5.4);`

c. `double z = m(4.5, 5.4);`

Key Point

6.9 The Scope of Variables

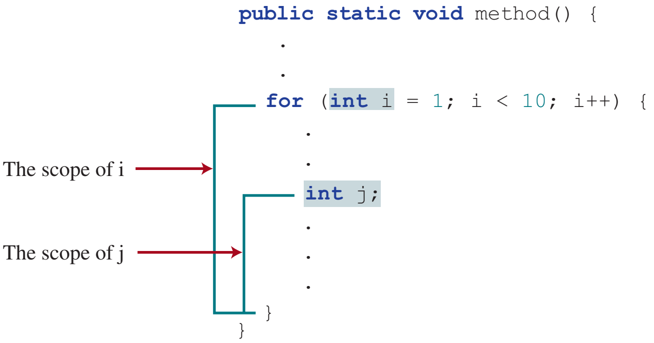
The scope of a variable is the part of the program where the variable can be referenced.

scope of variables

local variable

Section 2.5 introduced the scope of a variable. This section discusses the scope of variables in detail. A variable defined inside a method is referred to as a *local variable*. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared and assigned a value before it can be used.

A parameter is actually a local variable. The scope of a method parameter covers the entire method. A variable declared in the initial-action part of a **for**-loop header has its scope in the entire loop. However, a variable declared inside a **for**-loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable, as shown in Figure 6.5.



```
public static void method() {
  .
  .
  for (int i = 1; i < 10; i++) {
    .
    .
    int j;
    .
    .
    .
  }
}
```

FIGURE 6.5 A variable declared in the initial-action part of a **for**-loop header has its scope in the entire loop.

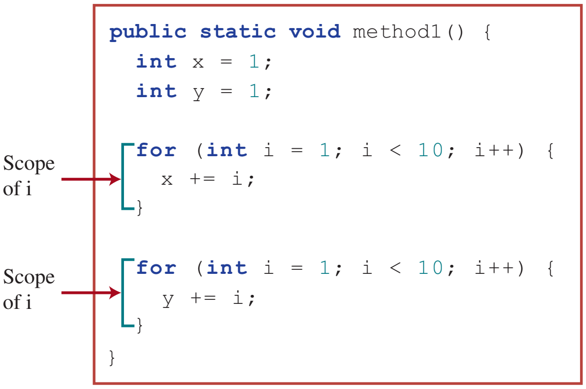
You can declare a local variable with the same name in different blocks in a method, but you cannot declare a local variable twice in the same block or in nested blocks, as shown in Figure 6.6.

Caution

A common mistake is to declare a variable in a **for** loop and then attempt to use it outside the loop. As shown in the following code, **i** is declared in the **for** loop, but it is accessed from the outside of the **for** loop, which causes a syntax error.

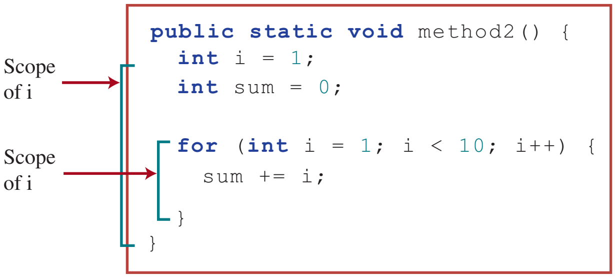
```
for (int i = 0; i < 10; i++) {
}
System.out.println(i); // Causes a syntax error on i
```

The last statement would cause a syntax error, because variable **i** is not defined outside of the **for** loop.



```
public static void method1() {
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++) {
    x += i;
  }

  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

(a) It is fine to declare i in two nonnested blocks



```
public static void method2() {
  int i = 1;
  int sum = 0;

  for (int i = 1; i < 10; i++) {
    sum += i;
  }
}
```

(b) It is wrong to declare i in two nested blocks

Figure 6.6 A variable can be declared multiple times in nonnested blocks, but only once in nested blocks.

Check Point

6.9.1 What is a local variable?

6.9.2 What is the scope of a local variable?

6.10 Case Study: Generating Random Characters

Key Point

A character is coded using an integer. Generating a random character is to generate an integer.

Computer programs process numerical data and characters. You have seen many examples that involve numerical data. It is also important to understand characters and how to process them. This section presents an example of generating random characters.

As introduced in Section 4.3, every character has a unique Unicode between `0` and `FFFF` in hexadecimal (`65535` in decimal). To generate a random character is to generate a random integer between `0` and `65535` using the following expression (note since `0 <= Math.random() < 1.0`, you have to add `1` to `65535`):

```
(int)(Math.random() * (65535 + 1))
```

Now let's consider how to generate a random lowercase letter. The Unicodes for lowercase letters are consecutive integers starting from the Unicode for `a`, then for `b`, `c`, . . ., and `z`. The Unicode for `a` is

```
(int)'a'
```

Thus, a random integer between `(int)'a'` and `(int)'z'` is

```
(int)((int)'a' + Math.random() * ((int)'z' – (int)'a' + 1))
```

As discussed in Section 4.3.3, all numeric operators can be applied to the `char` operands. The `char` operand is cast into a number if the other operand is a number or a character. Therefore, the preceding expression can be simplified as follows:

```
'a' + Math.random() * ('z' - 'a' + 1)
```

and a random lowercase letter is

```
(char)('a' + Math.random() * ('z' - 'a' + 1))
```

Hence, a random character between any two characters `ch1` and `ch2` with `ch1 < ch2` can be generated as follows:

```
(char)(ch1 + Math.random() * (ch2 - ch1 + 1))
```

This is a simple but useful discovery. Listing 6.10 defines a class named `RandomCharacter` with overloaded methods to get a certain type of random character. You can use these methods in your future projects.

LISTING 6.10 RandomCharacter.java

getRandomCharacter

getRandomLowerCaseLetter()

getRandomUpperCaseLetter()

getRandomDigitCharacter()

getRandomCharacter()

```
public class RandomCharacter {
  /** Generate a random character between ch1 and ch2 */
  public static char getRandomCharacter(char ch1, char ch2) {
    return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));
  }

  /** Generate a random lowercase letter */
  public static char getRandomLowerCaseLetter() {
    return getRandomCharacter('a', 'z');
  }

  /** Generate a random uppercase letter */
  public static char getRandomUpperCaseLetter() {
    return getRandomCharacter('A', 'Z');
  }

  /** Generate a random digit character */
  public static char getRandomDigitCharacter() {
    return getRandomCharacter('0', '9');
  }

  /** Generate a random character */
  public static char getRandomCharacter() {
    return getRandomCharacter('\u0000', '\uFFFF');
  }
}
```

Listing 6.11 gives a test program that displays 175 random lowercase letters.

LISTING 6.11 TestRandomCharacter.java

constants

lowercase letter

```
public class TestRandomCharacter {
  /** Main method */
  public static void main(String[] args) {
    final int NUMBER_OF_CHARS = 175;
    final int CHARS_PER_LINE = 25;

    // Print random characters between 'a' and 'z', 25 chars per line
    for (int i = 0; i < NUMBER_OF_CHARS; i++) {
      char ch = RandomCharacter.getRandomLowerCaseLetter();
      if ((i + 1) % CHARS_PER_LINE == 0)
        System.out.println(ch);
12       else
13         System.out.print(ch);
14     }
15   }
16 }
```

```
gmjsohezfkgtazqgmswfclrao
pnrunulnwmaztlfjedmpchcif
lalqdgivxkxpbzulrmqmbhikr
lbnrjlsopfxahssqhwuuljvbe
xbhdotzhpehbqmuwsfktwsoli
cbuwkzgxpmtzihgatdslvbwbz
bfesoklwbhnooygiigzdxuqni
```

Line 9 invokes **`getRandomLowerCaseLetter()`** defined in the **`RandomCharacter`** class. Note **`getRandomLowerCaseLetter()`** does not have any parameters, but you still have to use the parentheses when defining and invoking the method.

parentheses required

6.11 Method Abstraction and Stepwise Refinement

The key to developing software is to apply the concept of abstraction.

Key Point

VideoNote
Stepwise refinement

information hiding

method abstraction

You will learn many levels of abstraction from this book. *Method abstraction* is achieved by separating the use of a method from its implementation. The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is also known as *information hiding* or *encapsulation*. If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature. The implementation of the method is hidden from the client in a "black box," as shown in Figure 6.7.

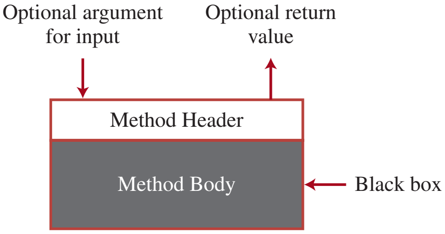


FIGURE 6.7 The method body can be thought of as a black box that contains the detailed implementation for the method.

You have already used the **`System.out.print`** method to display a string and the **`max`** method to find the maximum number. You know how to write the code to invoke these methods in your program, but as a user of these methods, you are not required to know how they are implemented.

The concept of method abstraction can be applied to the process of developing programs. When writing a large program, you can use the *divide-and-conquer* strategy, also known as *stepwise refinement*, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable problems.

divide and conquer

stepwise refinement

Suppose that you write a program that displays the calendar for a given month of the year. The program prompts the user to enter the year and the month, and then displays the entire calendar for the month, as presented in the following sample run:

```
Enter full year (e.g., 2012):  2012 ↵Enter
Enter month as number between 1 and 12:  3 ↵Enter

        March 2012
-----------------------------
 Sun Mon Tue Wed Thu Fri Sat
                   1   2   3
   4   5   6   7   8   9  10
  11  12  13  14  15  16  17
  18  19  20  21  22  23  24
  25  26  27  28  29  30
```

Let us use this example to demonstrate the divide-and-conquer approach.

6.11.1 Top-Down Design

How would you get started on such a program? Would you immediately start coding? Beginning programmers often start by trying to work out the solution to every detail. Although details are important in the final program, concern for detail in the early stages may block the problem-solving process. To make problem solving flow as smoothly as possible, this example begins by using method abstraction to isolate details from design and only later implements the details.

For this example, the problem is first broken into two subproblems: get input from the user, and print the calendar for the month. At this stage, you should be concerned with what the subproblems will achieve, not with how to get input and print the calendar for the month. You can draw a structure chart to help visualize the decomposition of the problem (see Figure 6.8a).

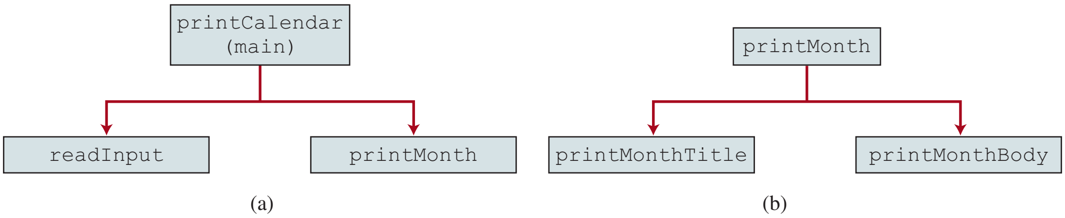


Figure 6.8 The structure chart shows the `printCalendar` problem is divided into two subproblems, `readInput` and `printMonth` in (a), and `printMonth` is divided into two smaller subproblems, `printMonthTitle` and `printMonthBody` in (b).

You can use `Scanner` to read input for the year and the month. The problem of printing the calendar for a given month can be broken into two subproblems: print the month title, and print the month body, as shown in Figure 6.8b. The month title consists of three lines: month and year, a dashed line, and the names of the seven days of the week. You need to get the month name (e.g., January) from the numeric month (e.g., 1). This is accomplished in `getMonthName` (see Figure 6.9a).

In order to print the month body, you need to know which day of the week is the first day of the month (`getStartDay`) and how many days the month has (`getNumberOfDaysInMonth`),

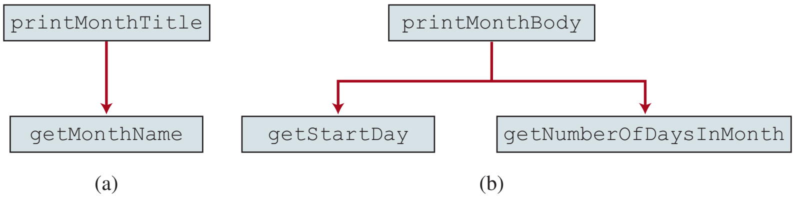


FIGURE 6.9 (a) To `printMonthTitle`, you need `getMonthName`. (b) The `printMonthBody` problem is refined into several smaller problems.

as shown in Figure 6.9b. For example, December 2013 has 31 days, and December 1, 2013 is a Sunday.

How would you get the start day for the first date in a month? There are several ways to do so. Assume you know that the start day for January 1, 1800 was a Wednesday (`START_DAY_FOR_JAN_1_1800 = 3`). You could compute the total number of days (`totalNumberOfDays`) between January 1, 1800 and the first date of the calendar month. The start day for the calendar month is `(totalNumberOfDays + START_DAY_FOR_JAN_1_1800) % 7`, since every week has seven days. Thus, the `getStartDay` problem can be further refined as `getTotalNumberOfDays`, as shown in Figure 6.10a.

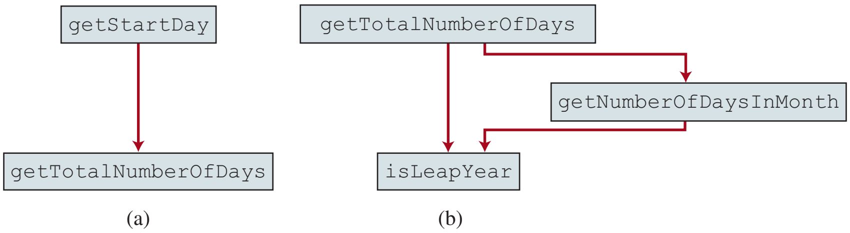


FIGURE 6.10 (a) To `getStartDay`, you need `getTotalNumberOfDays`. (b) The `getTotalNumberOfDays` problem is refined into two smaller problems.

To get the total number of days, you need to know whether the year is a leap year and the number of days in each month. Thus, `getTotalNumberOfDays` can be further refined into two subproblems: `isLeapYear` and `getNumberOfDaysInMonth`, as shown in Figure 6.10b. The complete structure chart is shown in Figure 6.11.

6.11.2 Top-Down and/or Bottom-Up Implementation

Now we turn our attention to implementation. In general, a subproblem corresponds to a method in the implementation, although some are so simple that this is unnecessary. You would need to decide which modules to implement as methods and which to combine with other methods. Decisions of this kind should be based on whether the overall program will be easier to read as a result of your choice. In this example, the subproblem `readInput` can be simply implemented in the `main` method.

top-down approach
stub

You can use either a "top-down" or a "bottom-up" approach. The top-down approach implements one method in the structure chart at a time from the top to the bottom. *Stubs*—a simple but incomplete version of a method—can be used for the methods waiting to be implemented. The use of stubs enables you to quickly build the framework of the program. Implement the `main` method first then use a stub for the `printMonth` method. For example,

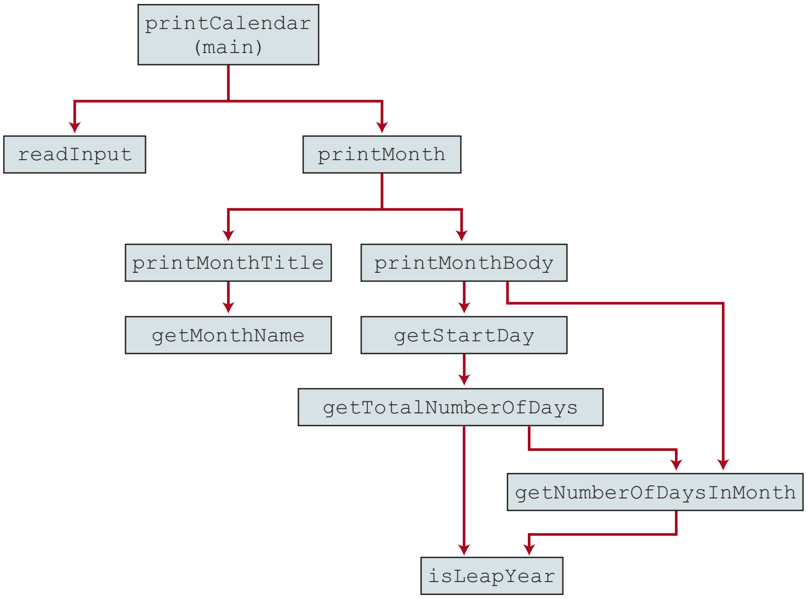

printCalendar (main)
readInput
printMonth
printMonthTitle
printMonthBody
getMonthName
getStartDay
getTotalNumberOfDays
getNumberOfDaysInMonth
isLeapYear

FIGURE 6.11 The structure chart shows the hierarchical relationship of the subproblems in the program.

let `printMonth` display the year and the month in the stub. Thus, your program may begin as follows:

```
public class PrintCalendar {
  /** Main method */
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter year
    System.out.print("Enter full year (e.g., 2012): ");
    int year = input.nextInt();

    // Prompt the user to enter month
    System.out.print("Enter month as a number between 1 and 12: ");
    int month = input.nextInt();

    // Print calendar for the month of the year
    printMonth(year, month);
  }

  /** A stub for printMonth may look like this */
  public static void printMonth(int year, int month) {
    System.out.print(month + " " + year);
  }

  /** A stub for printMonthTitle may look like this */
  public static void printMonthTitle(int year, int month) {
  }

  /** A stub for printMonthBody may look like this */
  public static void printMonthBody(int year, int month) {
  }
  /** A stub for getMonthName may look like this */
  public static String getMonthName(int month) {
    return "January"; // A dummy value
  }

  /** A stub for getStartDay may look like this */
  public static int getStartDay(int year, int month) {
    return 1; // A dummy value
  }

  /** A stub for getTotalNumberOfDays may look like this */
  public static int getTotalNumberOfDays(int year, int month) {
    return 10000; // A dummy value
  }

  /** A stub for getNumberOfDaysInMonth may look like this */
  public static int getNumberOfDaysInMonth(int year, int month) {
    return 31; // A dummy value
  }

  /** A stub for isLeapYear may look like this */
  public static boolean isLeapYear(int year) {
    return true; // A dummy value
  }
}
```

Compile and test the program, and fix any errors. You can now implement the `printMonth` method. For methods invoked from the `printMonth` method, you can again use stubs.

bottom-up approach

driver

The bottom-up approach implements one method in the structure chart at a time from the bottom to the top. For each method implemented, write a test program, known as the *driver*, to test it. The top-down and bottom-up approaches are equally good: Both approaches implement methods incrementally, help to isolate programming errors, and make debugging easy. They can be used together.

6.11.3 Implementation Details

The `isLeapYear(int year)` method can be implemented using the following code from Section 3.11:

```
return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);
```

Use the following facts to implement `getTotalNumberOfDaysInMonth(int year, int month)`:

- January, March, May, July, August, October, and December have 31 days.
- April, June, September, and November have 30 days.
- February has 28 days during a regular year, and 29 days during a leap year. A regular year, therefore, has 365 days, and a leap year has 366 days.

To implement `getTotalNumberOfDays(int year, int month)`, you need to compute the total number of days (`totalNumberOfDays`) between January 1, 1800 and the first day of the calendar month. You could find the total number of days between the year 1800 and the calendar year then figure out the total number of days prior to the calendar month in the calendar year. The sum of these two totals is `totalNumberOfDays`.

To print a body, first pad some space before the start day then print the lines for every week.

The complete program is given in Listing 6.12.

LISTING 6.12 PrintCalendar.java

printMonth

printMonthTitle

getMonthName

```
import java.util.Scanner;

public class PrintCalendar {
  /** Main method */
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter year
    System.out.print("Enter full year (e.g., 2012): ");
    int year = input.nextInt();

    // Prompt the user to enter month
    System.out.print("Enter month as a number between 1 and 12: ");
    int month = input.nextInt();

    // Print calendar for the month of the year
    printMonth(year, month);
  }

  /** Print the calendar for a month in a year */
  public static void printMonth(int year, int month) {
    // Print the headings of the calendar
    printMonthTitle(year, month);

    // Print the body of the calendar
    printMonthBody(year, month);
  }

  /** Print the month title, e.g., March 2012 */
  public static void printMonthTitle(int year, int month) {
    System.out.println("         " + getMonthName(month)
      + " " + year);
    System.out.println("-----------------------------");
    System.out.println(" Sun Mon Tue Wed Thu Fri Sat");
  }

  /** Get the English name for the month */
  public static String getMonthName(int month) {
    String monthName = "";
    switch (month) {
      case 1: monthName = "January"; break;
      case 2: monthName = "February"; break;
      case 3: monthName = "March"; break;
      case 4: monthName = "April"; break;
      case 5: monthName = "May"; break;
      case 6: monthName = "June"; break;
      case 7: monthName = "July"; break;
      case 8: monthName = "August"; break;
      case 9: monthName = "September"; break;
      case 10: monthName = "October"; break;
      case 11: monthName = "November"; break;
      case 12: monthName = "December";
    }

    return monthName;
  }

  /** Print month body */
```

printMonthBody

```
  public static void printMonthBody(int year, int month) {
    // Get start day of the week for the first date in the month
    int startDay = getStartDay(year, month);

    // Get number of days in the month
    int numberOfDaysInMonth = getNumberOfDaysInMonth(year, month);

    // Pad space before the first day of the month
    int i = 0;
    for (i = 0; i < startDay; i++)
      System.out.print("    ");

    for (i = 1; i <= numberOfDaysInMonth; i++) {
      System.out.printf("%4d", i);

      if ((i + startDay) % 7 == 0)
        System.out.println();
    }

    System.out.println();
  }

```

getStartDay

```
  /** Get the start day of month/1/year */
  public static int getStartDay(int year, int month) {
    final int START_DAY_FOR_JAN_1_1800 = 3;
    // Get total number of days from 1/1/1800 to month/1/year
    int totalNumberOfDays = getTotalNumberOfDays(year, month);

    // Return the start day for month/1/year
    return (totalNumberOfDays + START_DAY_FOR_JAN_1_1800) % 7;
  }

```

getTotalNumberOfDays

```
  /** Get the total number of days since January 1, 1800 */
  public static int getTotalNumberOfDays(int year, int month) {
    int total = 0;

    // Get the total days from 1800 to 1/1/year
    for (int i = 1800; i < year; i++)
      if (isLeapYear(i))
        total = total + 366;
      else
        total = total + 365;

    // Add days from Jan to the month prior to the calendar month
    for (int i = 1; i < month; i++)
      total = total + getNumberOfDaysInMonth(year, i);

    return total;
  }

```

getNumberOfDaysInMonth

```
  /** Get the number of days in a month */
  public static int getNumberOfDaysInMonth(int year, int month) {
    if (month == 1 || month == 3 || month == 5 || month == 7 ||
      month == 8 || month == 10 || month == 12)
      return 31;

    if (month == 4 || month == 6 || month == 9 || month == 11)
      return 30;

    if (month == 2) return isLeapYear(year)? 29: 28;
```

```

      return 0; // If month is incorrect
    }

    /** Determine if it is a leap year */
    public static boolean isLeapYear(int year) {
      return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);
    }
  }
```

isLeapYear

The program does not validate user input. For instance, if the user enters either a month not in the range between `1` and `12` or a year before `1800`, the program displays an erroneous calendar. To avoid this error, add an `if` statement to check the input before printing the calendar.

This program prints calendars for a month, but could easily be modified to print calendars for a whole year. Although it can print months only after January `1800`, it could be modified to print months before `1800`.

6.11.4 Benefits of Stepwise Refinement

Stepwise refinement breaks a large problem into smaller manageable subproblems. Each subproblem can be implemented using a method. This approach makes the program easier to write, reuse, debug, test, modify, and maintain.

Simpler Program

The print calendar program is long. Rather than writing a long sequence of statements in one method, stepwise refinement breaks it into smaller methods. This simplifies the program and makes the whole program easier to read and understand.

Reusing Methods

Stepwise refinement promotes code reuse within a program. The `isLeapYear` method is defined once and invoked from the `getTotalNumberOfDays` and `getNumberOfDaysInMonth` methods. This reduces redundant code.

Easier Developing, Debugging, and Testing

Since each subproblem is solved in a method, a method can be developed, debugged, and tested individually. This isolates the errors and makes developing, debugging, and testing easier.

When implementing a large program, use the top-down and/or bottom-up approach. Do not write the entire program at once. Using these approaches seems to take more development time (because you repeatedly compile and run the program), but it actually saves time and makes debugging easier.

incremental development and testing

Better Facilitating Teamwork

When a large problem is divided into subprograms, subproblems can be assigned to different programmers. This makes it easier for programmers to work in teams.

KEY TERMS

actual parameter	207
ambiguous invocation	223
argument	207
divide and conquer	227
formal parameter (i.e., parameter)	207
information hiding	227
method	206
method abstraction	227
method overloading	221
method signature	207
modifier	207
parameter	207
pass-by-value	214
scope of a variable	224
stepwise refinement	227
stub	229

CHAPTER SUMMARY

1. Making programs modular and reusable is one of the central goals in software engineering. Java provides many powerful constructs that help to achieve this goal. *Methods* are one such construct.

2. The method header specifies the *modifiers*, *return value type*, *method name*, and *parameters* of the method. The `static` modifier is used for all the methods in this chapter.

3. A method may return a value. The `returnValueType` is the data type of the value the method returns. If the method does not return a value, the `returnValueType` is the keyword `void`.

4. The *parameter list* refers to the type, order, and number of a method's parameters. The method name and the parameter list together constitute the *method signature.* Parameters are optional; that is, a method doesn't need to contain any parameters.

5. A return statement can also be used in a `void` method for terminating the method and returning to the method's caller. This is useful occasionally for circumventing the normal flow of control in a method.

6. The arguments that are passed to a method should have the same number, type, and order as the parameters in the method signature.

7. When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed, or when its method-ending closing brace is reached.

8. A value-returning method can also be invoked as a statement in Java. In this case, the caller simply ignores the return value.

9. A method can be overloaded. This means that two methods can have the same name, as long as their method parameter lists differ.

10. A variable declared in a method is called a local variable. The *scope of a local variable* starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared and initialized before it is used.

11. *Method abstraction* is achieved by separating the use of a method from its implementation. The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is known as *information hiding* or *encapsulation.*

12. Method abstraction modularizes programs in a neat, hierarchical manner. Programs written as collections of concise methods are easier to write, debug, maintain, and modify than would otherwise be the case. This writing style also promotes method reusability.

13. When implementing a large program, use the top-down and/or bottom-up coding approach. Do not write the entire program at once. This approach may seem to take more time for coding (because you are repeatedly compiling and running the program), but it actually saves time and makes debugging easier.

QUIZ

Answer the quiz for this chapter online at the Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

Note

A common error for the exercises in this chapter is that students don't implement the methods to meet the requirements even though the output from the main program is correct. For an example of this type of error, see liveexample.pearsoncmg.com/etc/CommonMethodErrorJava.pdf.

Sections 6.2–6.9

6.1 (*Math: pentagonal numbers*) A pentagonal number is defined as $n(3n-1)/2$ for $n = 1, 2, \ldots$, and so on. Therefore, the first few numbers are 1, 5, 12, 22, ... Write a method with the following header that returns a pentagonal number:

```java
public static int getPentagonalNumber(int n)
```

For example, `getPentagonalNumber(1)` returns `1` and `getPentagonalNumber(2)` returns `5`. Write a test program that uses this method to display the first 100 pentagonal numbers with 10 numbers on each line. Use the `%7d` format to display each number.

***6.2** (*Sum the digits in an integer*) Write a method that computes the sum of the digits in an integer. Use the following method header:

```java
public static int sumDigits(long n)
```

For example, `sumDigits(234)` returns `9` ($= 2 + 3 + 4$). (*Hint*: Use the `%` operator to extract digits and the `/` operator to remove the extracted digit. For instance, to extract 4 from 234, use `234 % 10` ($= 4$). To remove 4 from 234, use `234 / 10` ($= 23$). Use a loop to repeatedly extract and remove the digit until all the digits are extracted. Write a test program that prompts the user to enter an integer then displays the sum of all its digits.

****6.3** (*Palindrome integer*) Write the methods with the following headers:

```java
// Return the reversal of an integer, e.g., reverse(456) re-
turns 654
public static int reverse(int number)

// Return true if number is a palindrome
public static boolean isPalindrome(int number)
```

Use the `reverse` method to implement `isPalindrome`. A number is a palindrome if its reversal is the same as itself. Write a test program that prompts the user to enter an integer and reports whether the integer is a palindrome.

VideoNote
Reverse an integer

***6.4** (*Display an integer reversed*) Write a method with the following header to display an integer in reverse order:

```java
public static void reverse(int number)
```

For example, `reverse(3456)` displays `6543`. Write a test program that prompts the user to enter an integer then displays its reversal.

***6.5** (*Sort three numbers*) Write a method with the following header to display three numbers in increasing order:

```
public static void displaySortedNumbers(
  double num1, double num2, double num3)
```

Write a test program that prompts the user to enter three numbers and invokes the method to display them in increasing order.

***6.6** (*Display patterns*) Write a method to display a pattern as follows:

```
          1
        2 1
      3 2 1
...
n n-1 ... 3 2 1
```

The method header is

```
public static void displayPattern(int n)
```

Write a test program that prompts the user to enter a number n and invokes `displayPattern(n)` to display the pattern.

***6.7** (*Financial application: compute the future investment value*) Write a method that computes future investment value at a given interest rate for a specified number of years. The future investment is determined using the formula in Programming Exercise 2.21.

Use the following method header:

```
public static double futureInvestmentValue(
  double investmentAmount, double monthlyInterestRate,int years)
```

For example, `futureInvestmentValue(10000, 0.05/12, 5)` returns `12833.59`.

Write a test program that prompts the user to enter the investment amount (e.g., 1,000) and the interest rate (e.g., 9%) and prints a table that displays future value for the years from 1 to 30, as shown below:

```
The amount invested: 1000 ↵Enter
Annual interest rate: 9 ↵Enter
Years     Future Value
1            1093.80
2            1196.41
...
29          13467.25
30          14730.57
```

6.8 (*Conversions between Celsius and Fahrenheit*) Write a class that contains the following two methods:

```
/** Convert from Celsius to Fahrenheit */
public static double celsiusToFahrenheit(double celsius)

/** Convert from Fahrenheit to Celsius */
public static double fahrenheitToCelsius(double fahrenheit)
```

The formula for the conversion is as follows:

```
fahrenheit = (9.0 / 5) * celsius + 32
celsius = (5.0 / 9) * (fahrenheit – 32)
```

Write a test program that invokes these methods to display the following table:

```
Celsius    Fahrenheit   |     Fahrenheit    Celsius
-----------------------------------------------------
40.0       104.0        |     120.0         48.89
39.0       102.2        |     110.0         43.33
38.0       100.4        |     100.0         37.78
37.0       98.6         |     90.0          32.22
36.0       96.8         |     80.0          26.67
35.0       95.0         |     70.0          21.11
34.0       93.2         |     60.0          21.11
33.0       91.4         |     50.0          10.00
32.0       89.6         |     40.0          4.44
31.0       87.8         |     30.0          -1.11
```

6.9 (*Conversions between feet and meters*) Write a class that contains the following two methods:

```
/** Convert from feet to meters */
public static double footToMeter(double foot)

/** Convert from meters to feet */
public static double meterToFoot(double meter)
```

The formula for the conversion is:

```
meter = 0.305 * foot
foot = 3.279 * meter
```

Write a test program that invokes these methods to display the following tables:

```
Feet       Meters       |     Meters        Feet
- - - - - - - - - - - - - - - - - - - - - - - - - -
1.0        0.305        |     20.0          65.574
2.0        0.610        |     25.0          81.967
3.0        0.915        |     30.0          98.361
4.0        1.220        |     35.0          114.754
5.0        1.525        |     40.0          131.148
6.0        1.830        |     45.0          147.541
7.0        2.135        |     50.0          163.934
8.0        2.440        |     55.0          180.328
9.0        2.745        |     60.0          196.721
10.0       3.050        |     65.0          213.115
```

6.10 (*Use the* `isPrime` *Method*) Listing 6.7, PrimeNumberMethod.java, provides the `isPrime(int number)` method for testing whether a number is prime. Use this method to find the number of prime numbers less than `10000`.

6.11 (*Financial application: compute commissions*) Write a method that computes the commission, using the scheme in Programming Exercise 5.39. The header of the method is as follows:

```
public static double computeCommission(double salesAmount)
```

Write a test program that displays the following table:

```
Sales Amount       Commission
10000              900.0
15000              1500.0
20000              2100.0
25000              2700.0
30000              3300.0
35000              3900.0
40000              4500.0
45000              5100.0
50000              5700.0
55000              6300.0
60000              6900.0
65000              7500.0
70000              8100.0
75000              8700.0
80000              9300.0
85000              9900.0
90000              10500.0
95000              11100.0
100000             11700.0
```

6.12 (*Display characters*) Write a method that prints characters using the following header:

```
public static void printChars(char ch1, char ch2, int
  numberPerLine)
```

This method prints the characters between `ch1` and `ch2` with the specified numbers per line. Write a test program that prints 10 characters per line from `1` to `Z`. Characters are separated by exactly one space.

***6.13** (*Sum series*) Write a method to compute the following summation:

$$m(i) = \frac{1}{2} + \frac{2}{3} + \cdots + \frac{i}{i+1}$$

Write a test program that displays the following table:

```
i                  m(i)
1                  0.5000
2                  1.1667
3                  1.9167
```

```
i                       m(i)
4                       2.7167
5                       3.5500
6                       4.4071
7                       5.2821
8                       6.1710
9                       7.0710
10                      7.9801
11                      8.8968
12                      9.8199
13                      10.7484
14                      11.6818
15                      12.6193
16                      13.5604
17                      14.5049
18                      15.4523
19                      16.4023
20                      17.3546
```

VideoNote
Estimate π

***6.14** (*Estimate* π) π can be computed using the following summation:

$$m(i) = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots + \frac{(-1)^{i+1}}{2i - 1}\right)$$

Write a method that returns `m(i)` for a given `i` and write a test program that displays the following table:

```
i                       m(i)
1                       4.0000
101                     3.1515
201                     3.1466
301                     3.1449
401                     3.1441
501                     3.1436
601                     3.1433
701                     3.1430
801                     3.1428
901                     3.1427
```

***6.15** (*Financial application: print a tax table*) Listing 3.5 gives a program to compute tax. Write a method for computing tax using the following header:

```
public static double computeTax(int status, double
  taxableIncome)
```

Use this method to write a program that prints a tax table for taxable income from $50,000 to $60,000 with intervals of $50 for all the following statuses:

```
Taxable   Single    Married Joint   Married     Head of
Income              or Qualifying   Separate    House hold
                    Widow(er)
50000     8688      6665            8688        7353
50050     8700      6673            8700        7365
50100     8712      6680            8712        7378
50150     8725      6688            8725        7390
. . .
59850     11150     8142            11150       9815
59900     11162     8150            11162       9828
59950     11175     8158            11175       9840
60000     11188     8165            11188       9853
```

Hint: round the tax into integers using `Math.round` (i.e., `Math .round(computeTax(status, taxableIncome))`).

***6.16** (*Number of days in a year*) Write a method that returns the number of days in a year using the following header:

```
public static int numberOfDaysInAYear(int year)
```

Write a test program that displays the number of days in year from 2000 to 2020.

Sections 6.10 and 6.11

***6.17** (*Display matrix of 0s and 1s*) Write a method that displays an *n*-by-*n* matrix using the following header: Here is a sample run:

```
public static int printMatrix(int year)
```

that prompts the user to enter `n` and displays an *n*-by-*n* matrix. Here is a sample run:

```
Enter n: 3 ↵Enter
0 1 0
0 0 0
1 1 1
```

```
public static void printMatrix(int n)
```

Each element is 0 or 1, which is generated randomly. Write a test program that prompts the user to enter `n` and displays an *n*-by-*n* matrix.

****6.18** (*Check password*) Some Websites impose certain rules for passwords. Write a method that checks whether a string is a valid password. Suppose the password rules are as follows:

- A password must have at least eight characters.
- A password must contain only letters and digits.
- A password must contain at least two digits.

Write a program that prompts the user to enter a password and displays `Valid Password` if the rules are followed, or `Invalid Password` otherwise.

***6.19** (*Triangles*) Implement the following two methods:

```
/** Return true if the sum of every two sides is
 * greater than the third side. */
public static boolean isValid(
  double side1, double side2, double side3)

/** Return the area of the triangle. */
public static double area(
  double side1, double side2, double side3)
```

Write a test program that reads three sides for a triangle and uses the **`isValid`** method to test if the input is valid and uses the **`area`** method to obtain the area. The program displays the area if the input is valid. Otherwise, it displays that the input is invalid. The formula for computing the area of a triangle is given in Programming Exercise 2.19.

***6.20** (*Count the letters in a string*) Write a method that counts the number of letters in a string using the following header:

```
public static int countLetters(String s)
```

Write a test program that prompts the user to enter a string and displays the number of letters in the string.

***6.21** (*Phone keypads*) The international standard letter/number mapping for telephones is given in Programming Exercise 4.15. Write a method that returns a number, given an uppercase letter, as follows:

```
public static int getNumber(char uppercaseLetter)
```

Write a test program that prompts the user to enter a phone number as a string. The input number may contain letters. The program translates a letter (uppercase or lowercase) to a digit and leaves all other characters intact. Here are sample runs of the program:

```
Enter a string: 1-800-Flowers ↵Enter
1-800-3569377
```

```
Enter a string: 1800flowers ↵Enter
18003569377
```

****6.22** (*Math: approximate the square root*) There are several techniques for implementing the **`sqrt`** method in the **`Math`** class. One such technique is known as the *Babylonian method*. It approximates the square root of a number, **`n`**, by repeatedly performing the calculation using the following formula:

```
nextGuess = (lastGuess + n / lastGuess) / 2
```

When **`nextGuess`** and **`lastGuess`** are almost identical, **`nextGuess`** is the approximated square root. The initial guess can be any positive value (e.g., **`1`**). This value will be the starting value for **`lastGuess`**. If the difference between **`nextGuess`** and **`lastGuess`** is less than a very small number, such as **`0.0001`**, you can claim that **`nextGuess`** is the approximated square root of **`n`**. If not, **`nextGuess`** becomes **`lastGuess`** and the approximation process continues. Implement the following method that returns the square root of **`n`**:

```
public static double sqrt(long n)
```

Write a test program that prompts the user to enter a positive double value and displays its square root.

***6.23** (*Occurrences of a specified character*) Write a method that finds the number of occurrences of a specified character in a string using the following header:

```
public static int count(String str, char a)
```

For example, `count("Welcome", 'e')` returns `2`. Write a test program that prompts the user to enter a string followed by a character then displays the number of occurrences of the character in the string.

Sections 6.10–6.12

****6.24** (*Display current date and time*) Listing 2.7, ShowCurrentTime.java, displays the current time. Revise this example to display the current date and time. The calendar example in Listing 6.12, PrintCalendar.java, should give you some ideas on how to find the year, month, and day.

****6.25** (*Convert milliseconds to hours, minutes, and seconds*) Write a method that converts milliseconds to hours, minutes, and seconds using the following header:

```
public static String convertMillis(long millis)
```

The method returns a string as *hours:minutes:seconds*. For example, `convertMillis(5500)` returns a string `0:0:5`, `convertMillis(100000)` returns a string `0:1:40`, and `convertMillis(555550000)` returns a string `154:19:10`. Write a test program that prompts the user to enter a long integer for milliseconds and displays a string in the format of hours:minutes:seconds.

Comprehensive

****6.26** (*Palindromic prime*) A *palindromic prime* is a prime number and also palindromic. For example, 131 is a prime and also a palindromic prime, as are 313 and 757. Write a program that displays the first 100 palindromic prime numbers. Display 10 numbers per line, separated by exactly one space, as follows:

```
2 3 5 7 11 101 131 151 181 191
313 353 373 383 727 757 787 797 919 929
...
```

****6.27** (*Emirp*) An *emirp* (prime spelled backward) is a nonpalindromic prime number whose reversal is also a prime. For example, 17 is a prime and 71 is a prime, so 17 and 71 are emirps. Write a program that displays the first 100 emirps. Display 10 numbers per line, separated by exactly one space, as follows:

```
13 17 31 37 71 73 79 97 107 113
149 157 167 179 199 311 337 347 359 389
...
```

****6.28** (*Mersenne prime*) A prime number is called a *Mersenne prime* if it can be written in the form $2^p - 1$ for some positive integer p. Write a program that finds all Mersenne primes with $p \leq 31$ and displays the output as follows:

```
p          2^p – 1
-------------------
2          3
3          7
5          31
...
```

****6.29** (*Twin primes*) Twin primes are a pair of prime numbers that differ by 2. For example, 3 and 5 are twin primes, 5 and 7 are twin primes, and 11 and 13 are twin primes. Write a program to find all twin primes less than 1,000. Display the output as follows:

```
(3, 5)
(5, 7)
...
```

****6.30** (*Game: craps*) Craps is a popular dice game played in casinos. Write a program to play a variation of the game, as follows:

Roll two dice. Each die has six faces representing values 1, 2, . . ., and 6, respectively. Check the sum of the two dice. If the sum is 2, 3, or 12 (called *craps*), you lose; if the sum is 7 or 11 (called *natural*), you win; if the sum is another value (i.e., 4, 5, 6, 8, 9, or 10), a point is established. Continue to roll the dice until either a 7 or the same point value is rolled. If 7 is rolled, you lose. Otherwise, you win.

Your program acts as a single player. Here are some sample runs.

```
You rolled 5 + 6 = 11
You win
```

```
You rolled 1 + 2 = 3
You lose
```

```
You rolled 4 + 4 = 8
point is 8
You rolled 6 + 2 = 8
You win
```

```
You rolled 3 + 2 = 5
point is 5
You rolled 2 + 5 = 7
You lose
```

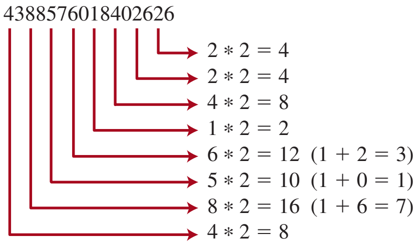
****6.31** (*Financial: credit card number validation*) Credit card numbers follow certain patterns. A credit card number must have between 13 and 16 digits. It must start with

- 4 for Visa cards
- 5 for Master cards
- 37 for American Express cards
- 6 for Discover cards

In 1954, Hans Luhn of IBM proposed an algorithm for validating credit card numbers. The algorithm is useful to determine whether a card number is entered correctly, or whether a credit card is scanned correctly by a scanner. Credit card numbers are generated following this validity check, commonly known as the

Luhn check or the *Mod 10 check,* which can be described as follows (for illustration, consider the card number 4388576018402626):

1. Double every second digit from right to left. If doubling of a digit results in a two-digit number, add up the two digits to get a single-digit number.



2. Now add all single-digit numbers from Step 1.

$$4 + 4 + 8 + 2 + 3 + 1 + 7 + 8 = 37$$

3. Add all digits in the odd places from right to left in the card number.

$$6 + 6 + 0 + 8 + 0 + 7 + 8 + 3 = 38$$

4. Sum the results from Step 2 and Step 3.

$$37 + 38 = 75$$

5. If the result from Step 4 is divisible by 10, the card number is valid; otherwise, it is invalid. For example, the number 4388576018402626 is invalid, but the number 4388576018410707 is valid.

Write a program that prompts the user to enter a credit card number as a `long` integer. Display whether the number is valid or invalid. Design your program to use the following methods:

```
/** Return true if the card number is valid */
public static boolean isValid(long number)

/** Get the result from Step 2 */
public static int sumOfDoubleEvenPlace(long number)

/** Return this number if it is a single digit, otherwise,
 * return the sum of the two digits */
public static int getDigit(int number)

/** Return sum of odd-place digits in number */
public static int sumOfOddPlace(long number)

/** Return true if the number d is a prefix for number */
public static boolean prefixMatched(long number, int d)

/** Return the number of digits in d */
public static int getSize(long d)

/** Return the first k number of digits from number. If the
 * number of digits in number is less than k, return number. */
public static long getPrefix(long number, int k)
```

(You may also implement this program by reading the input as a string and processing the string to validate the credit card.)

```
Enter a credit card number as a long integer:
  4388576018410707 ↵Enter
4388576018410707 is valid
```

```
Enter a credit card number as a long integer:
  4388576018402626 ↵Enter
4388576018402626 is invalid
```

****6.32** (*Game: chance of winning at craps*) Revise Programming Exercise 6.30 to run it 10,000 times and display the number of winning games.

****6.33** (*Current date and time*) Invoking `System.currentTimeMillis()` returns the elapsed time in milliseconds since midnight of January 1, 1970. Write a program that displays the date and time. Here is a sample run:

```
Current date and time is May 16, 2012 10:34:23
```

****6.34** (*Print calendar*) Programming Exercise 3.21 uses Zeller's congruence to calculate the day of the week. Simplify Listing 6.12, PrintCalendar.java, using Zeller's algorithm to get the start day of the month.

6.35 (*Geometry: area of a pentagon*) The area of a pentagon can be computed using the following formula:

$$Area = \frac{5 \times s^2}{4 \times \tan\left(\frac{\pi}{5}\right)}$$

Write a method that returns the area of a pentagon using the following header:

`public static double area(double side)`

Write a main method that prompts the user to enter the side of a pentagon and displays its area. Here is a sample run:

```
Enter the side: 5.5 ↵Enter
The area of the pentagon is 52.04444136781625
```

***6.36** (*Geometry: area of a regular polygon*) A regular polygon is an *n*-sided polygon in which all sides are of the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). The formula for computing the area of a regular polygon is

$$Area = \frac{n \times s^2}{4 \times \tan\left(\frac{\pi}{n}\right)}$$

Write a method that returns the area of a regular polygon using the following header:

```
public static double area(int n, double side)
```

Write a main method that prompts the user to enter the number of sides and the side of a regular polygon and displays its area. Here is a sample run:

```
Enter the number of sides: 5 ↵Enter
Enter the side: 6.5 ↵Enter
The area of the polygon is 72.69017017488385
```

6.37 (*Format an integer*) Write a method with the following header to format the integer with the specified width.

```
public static String format(int number, int width)
```

The method returns a string for the number with one or more prefix `0`s. The size of the string is the width. For example, `format(34, 4)` returns `0034` and `format(34, 5)` returns `00034`. If the number is longer than the width, the method returns the string representation for the number. For example, `format(34, 1)` returns `34`.

Write a test program that prompts the user to enter a number and its width, and displays a string returned by invoking `format(number, width)`.

***6.38** (*Generate random characters*) Use the methods in `RandomCharacter` in Listing 6.10 to print 100 uppercase letters then 100 single digits, printing 50 per line.

6.39 (*Geometry: point position*) Programming Exercise 3.32 shows how to test whether a point is on the left side of a directed line, on the right, or on the same line. Write the methods with the following headers:

```
/** Return true if point (x2, y2) is on the left side of the
 * directed line from (x0, y0) to (x1, y1) */
public static boolean leftOfTheLine(double x0, double y0,
  double x1, double y1, double x2, double y2)

/** Return true if point (x2, y2) is on the same
 * line from (x0, y0) to (x1, y1) */
public static boolean onTheSameLine(double x0, double y0,
  double x1, double y1, double x2, double y2)

/** Return true if point (x2, y2) is on the
 * line segment from (x0, y0) to (x1, y1) */
public static boolean onTheLineSegment(double x0, double y0,
  double x1, double y1, double x2, double y2)
```

Write a program that prompts the user to enter the three points for **p0**, **p1**, and **p2** and displays whether **p2** is on the left side of the line from **p0** to **p1**, right side, the same line, or on the line segment. Here are some sample runs:

```
Enter three points for p0, p1, and p2: 1 1 2 2 1.5 1.5 ↵Enter
(1.5, 1.5) is on the line segment from (1.0, 1.0) to (2.0, 2.0)
```

```
Enter three points for p0, p1, and p2: 1 1 2 2 3 3 ↵Enter
(3.0, 3.0) is on the same line from (1.0, 1.0) to (2.0, 2.0)
```

```
Enter three points for p0, p1, and p2: 1 1 2 2 1 1.5 ↵Enter
(1.0, 1.5) is on the left side of the line
  from (1.0, 1.0) to (2.0, 2.0)
```

```
Enter three points for p0, p1, and p2: 1 1 2 2 1 -1 ↵Enter
(1.0, -1.0) is on the right side of the line
  from (1.0, 1.0) to (2.0, 2.0)
```

CHAPTER

7

SINGLE-DIMENSIONAL ARRAYS

Objectives

- To describe why arrays are necessary in programming (§7.1).
- To declare array reference variables and create arrays (§§7.2.1 and 7.2.2).
- To obtain array size using `arrayRefVar.length` and know default values in an array (§7.2.3).
- To access array elements using indexes (§7.2.4).
- To declare, create, and initialize an array using an array initializer (§7.2.5).
- To program common array operations (displaying arrays, summing all elements, finding the minimum and maximum elements, random shuffling, and shifting elements) (§7.2.6).
- To simplify programming using the foreach loops (§7.2.7).
- To apply arrays in application development (`AnalyzeNumbers`, and `DeckOfCards`) (§§7.3 and 7.4).
- To copy contents from one array to another (§7.5).
- To develop and invoke methods with array arguments and return values (§§7.6–7.8).
- To define a method with a variable-length argument list (§7.9).
- To search elements using the linear (§7.10.1) or binary (§7.10.2) search algorithm.
- To sort an array using the selection sort approach (§7.11).
- To use the methods in the `java.util.Arrays` class (§7.12).
- To pass arguments to the main method from the command line (§7.13).

7.1 Introduction

A single array variable can reference a large collection of data.

Key Point

problem

why array?

Often you will have to store a large number of values during the execution of a program. Suppose, for instance, that you need to read 100 numbers, compute their average, and find out how many numbers are above the average. Your program first reads the numbers and computes their average, then compares each number with the average to determine whether it is above the average. In order to accomplish this task, the numbers must all be stored in variables. You have to declare 100 variables and repeatedly write almost identical code 100 times. Writing a program this way would be impractical. So, how do you solve this problem?

An efficient, organized approach is needed. Java and most other high-level languages provide a data structure, the *array*, which stores a fixed-size sequential collection of elements of the same type. In the present case, you can store all 100 numbers into an array and access them through a single array variable.

This chapter introduces single-dimensional arrays. The next chapter will introduce two-dimensional and multidimensional arrays.

7.2 Array Basics

Once an array is created, its size is fixed. An array reference variable is used to access the elements in an array using an index.

Key Point

index

An array is used to store a collection of data, but often we find it more useful to think of an array as a collection of variables of the same type. Instead of declaring individual variables, such as **`number0`**, **`number1`**, . . . , and **`number99`**, you declare one array variable such as **`numbers`** and use **`numbers[0]`**, **`numbers[1]`**, . . . , and **`numbers[99]`** to represent individual variables. This section introduces how to declare array variables, create arrays, and process arrays using indexes.

7.2.1 Declaring Array Variables

element type

To use an array in a program, you must declare a variable to reference the array and specify the array's *element type*. Here is the syntax for declaring an array variable.

```
elementType[] arrayRefVar;
```

or

```
elementType arrayRefVar[]; // Allowed, but not preferred
```

The **`elementType`** can be any data type, and all elements in the array will have the same data type. For example, the following code declares a variable **`myList`** that references an array of double elements.

```
double[] myList;
```

or

```
double myList[]; // Allowed, but not preferred
```

Note

You can use **`elementType arrayRefVar[]`** to declare an array variable. This style comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers. The style **`elementType[] arrayRefVar`** is preferred.

preferred syntax

7.2.2 Creating Arrays

Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. It creates only a storage location for the reference to an array. If a variable does not contain a reference to an array, the value of the variable is `null`. You cannot assign elements to an array unless it has already been created. After an array variable is declared, you can create an array by using the `new` operator and assign its reference to the variable with the following syntax:

`null`

```
arrayRefVar = new elementType[arraySize];
```

new operator

This statement does two things: (1) it creates an array using `new elementType[arraySize]` and (2) it assigns the reference of the newly created array to the variable `arrayRefVar`.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement as

```
elementType[] arrayRefVar = new elementType[arraySize];
```

or

```
elementType arrayRefVar[] = new elementType[arraySize];
```

Here is an example of such a statement:

```
double[] myList = new double[10];
```

This statement declares an array variable, `myList`, creates an array of 10 elements of `double` type, and assigns its reference to `myList`. To assign values to the elements, use the syntax

```
arrayRefVar[index] = value;
```

For example, the following code initializes the array:

```
myList[0] = 5.6;
myList[1] = 4.5;
myList[2] = 3.3;
myList[3] = 13.2;
myList[4] = 4.0;
myList[5] = 34.33;
myList[6] = 34.0;
myList[7] = 45.45;
myList[8] = 99.993;
myList[9] = 11123;
```

This array is illustrated in Figure 7.1.

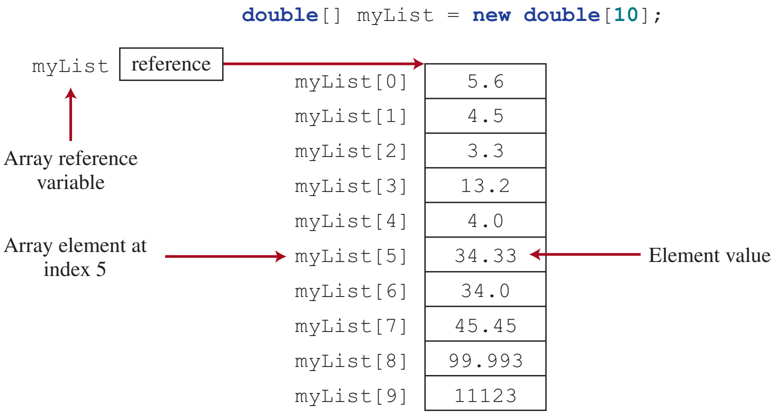

double[] myList = new double[10];
myList
reference
Array reference variable
Array element at index 5
Element value
myList[0] 5.6
myList[1] 4.5
myList[2] 3.3
myList[3] 13.2
myList[4] 4.0
myList[5] 34.33
myList[6] 34.0
myList[7] 45.45
myList[8] 99.993
myList[9] 11123

FIGURE 7.1 The array `myList` has 10 elements of `double` type and `int` indices from `0` to `9`.

array vs. array variable

Note

An array variable that appears to hold an array actually contains a reference to that array. Strictly speaking, an array variable and an array are different, but most of the time the distinction can be ignored. Thus, it is all right to say, for simplicity, that `myList` is an array, instead of stating, at greater length, that `myList` is a variable that contains a reference to an array of double elements.

7.2.3 Array Size and Default Values

array length

When space for an array is allocated, the array size must be given, specifying the number of elements that can be stored in it. The size of an array cannot be changed after the array is created. Size can be obtained using `arrayRefVar.length`. For example, `myList.length` is `10`.

default values

When an array is created, its elements are assigned the default value of `0` for the numeric primitive data types, `\u0000` for `char` types, and `false` for `boolean` types.

7.2.4 Accessing Array Elements

0 based index

The array elements are accessed through the index. Array indices are `0` based; that is, they range from `0` to `arrayRefVar.length - 1`. In the example in Figure 7.1, `myList` holds 10 `double` values, and the indices are from `0` to `9`.

indexed variable

Each element in the array is represented using the following syntax, known as an *indexed variable*:

```
arrayRefVar[index];
```

For example, `myList[9]` represents the last element in the array `myList`.

Caution

Some programming languages use parentheses to reference an array element, as in `myList(9)`, but Java uses brackets, as in `myList[9]`.

An indexed variable can be used in the same way as a regular variable. For example, the following code adds the values in `myList[0]` and `myList[1]` to `myList[2]`:

```
myList[2] = myList[0] + myList[1];
```

The following loop assigns `0` to `myList[0]`, `1` to `myList[1]`, . . . , and `9` to `myList[9]`:

```
for (int i = 0; i < myList.length; i++) {
  myList[i] = i;
}
```

7.2.5 Array Initializers

array initializer

Java has a shorthand notation, known as the *array initializer*, which combines the declaration, creation, and initialization of an array in one statement using the following syntax:

```
elementType[] arrayRefVar = {value0, value1, ..., valuek};
```

For example, the statement

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

declares, creates, and initializes the array `myList` with four elements, which is equivalent to the following statements:

```
double[] myList = new double[4];
myList[0] = 1.9;
myList[1] = 2.9;
```

```
myList[2] = 3.4;
myList[3] = 3.5;
```

Caution

The `new` operator is not used in the array-initializer syntax. Using an array initializer, you have to declare, create, and initialize the array all in one statement. Splitting it would cause a syntax error. Thus, the next statement is wrong:

```
double[] myList;
myList = {1.9, 2.9, 3.4, 3.5}; // Wrong
```

7.2.6 Processing Arrays

When processing array elements, you will often use a `for` loop for one of two reasons:

1. All of the elements in an array are of the same type. They are evenly processed in the same fashion repeatedly using a loop.
2. Since the size of the array is known, it is natural to use a `for` loop.

Assume that the array is created as follows:

```
double[] myList = new double[10];
```

The following are some examples of processing arrays:

1. *Initializing arrays with input values:* The following loop initializes the array `myList` with user input values:

```
java.util.Scanner input = new java.util.Scanner(System.in);
System.out.print("Enter " + myList.length + " values: ");
for (int i = 0; i < myList.length; i++)
  myList[i] = input.nextDouble();
```

2. *Initializing arrays with random values:* The following loop initializes the array `myList` with random values between `0.0` and `100.0`, but less than `100.0`:

```
for (int i = 0; i < myList.length; i++) {
  myList[i] = Math.random() * 100;
}
```

3. *Displaying arrays:* To print an array, you have to print each element in the array using a loop such as the following:

```
for (int i = 0; i < myList.length; i++) {
  System.out.print(myList[i] + " ");
}
```

Tip

print character array

For an array of the `char[]` type, it can be printed using one print statement. For example, the following code displays `Dallas`:

```
char[] city = {'D', 'a', 'l', 'l', 'a', 's'};
System.out.println(city);
```

4. *Summing all elements:* Use a variable named `total` to store the sum. Initially `total` is `0`. Add each element in the array to `total` using a loop such as the following:

```
double total = 0;
for (int i = 0; i < myList.length; i++) {
  total += myList[i];
}
```

5. *Finding the largest element:* Use a variable named `max` to store the largest element. Initially `max` is `myList[0]`. To find the largest element in the array `myList`, compare each element with `max`, and update `max` if the element is greater than `max`.

```
double max = myList[0];
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) max = myList[i];
}
```

6. *Finding the smallest index of the largest element:* Often you need to locate the largest element in an array. If an array has multiple elements with the same largest value, find the smallest index of such an element. Suppose that the array `myList` is {`1`, `5`, `3`, `4`, `5`, `5`}. The largest element is `5`, and the smallest index for `5` is `1`. Use a variable named `max` to store the largest element, and a variable named `indexOfMax` to denote the index of the largest element. Initially `max` is `myList[0]` and `indexOfMax` is `0`. Compare each element in `myList` with `max` and update `max` and `indexOfMax` if the element is greater than `max`.

```
double max = myList[0];
int indexOfMax = 0;
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) {
    max = myList[i];
    indexOfMax = i;
  }
}
```

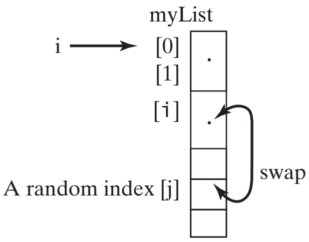
random shuffling

VideoNote

Random shuffling

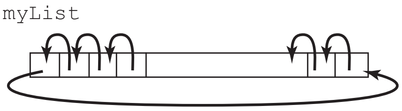
7. *Random shuffling:* In many applications, you need to randomly reorder the elements in an array. This is called *shuffling*. To accomplish this, for each element `myList[i]`, randomly generate an index `j` and swap `myList[i]` with `myList[j]`, as follows:

```
for (int i = 0; i < myList.length – 1; i++) {
  // Generate an index j randomly
  int j = (int)(Math.random()
    * myList.length);

  // Swap myList[i] with myList[j]
  double temp = myList[i];
  myList[i] = myList[j];
  myList[j] = temp;
}
```



8. *Shifting elements:* Sometimes you need to shift the elements left or right. Here is an example of shifting the elements one position to the left and filling the last element with the first element:

```
double temp = myList[0]; // Retain the first element

// Shift elements left
for (int i = 1; i < myList.length; i++) {
   myList[i - 1] = myList[i];
}

// Move the first element to fill in the last position
myList[myList.length - 1] = temp;
```



9. *Simplifying coding:* Arrays can be used to greatly simplify coding for certain tasks. For example, suppose you wish to obtain the English name of a given month by its number. If the month names are stored in an array, the month name for a given month can be accessed simply via

the index. The following code prompts the user to enter a month number and displays its month name:

```
String[] months = {"January", "February",..., "December"};
System.out.print("Enter a month number (1 to 12): ");
int monthNumber = input.nextInt();
System.out.println("The month is " + months[monthNumber - 1]);
```

If you didn't use the `months` array, you would have to determine the month name using a lengthy multiway `if-else` statement as follows:

```
if (monthNumber == 1)
  System.out.println("The month is January");
else if (monthNumber == 2)
  System.out.println("The month is February");
...
else
  System.out.println("The month is December");
```

7.2.7 Foreach Loops

Java supports a convenient `for` loop, known as a *foreach loop*, which enables you to traverse the array sequentially without using an index variable. For example, the following code displays all the elements in the array `myList`:

```
for (double e: myList) {
  System.out.println(e);
}
```

You can read the code as "for each element `e` in `myList`, do the following." Note that the variable, `e`, must be declared as the same type as the elements in `myList`.

In general, the syntax for a foreach loop is

```
for (elementType element: arrayRefVar) {
  // Process the element
}
```

You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

Caution

Accessing an array out of bounds is a common programming error that throws a runtime `ArrayIndexOutOfBoundsException`. To avoid it, make sure you do not use an index beyond `arrayRefVar.length – 1` or simply using a foreach loop if possible.

ArrayIndexOutOfBounds-Exception

Programmers often mistakenly reference the first element in an array with index `1`, but it should be `0`. This is called the *off-by-one error*. Another common off-by-one error in a loop is using `<=` where `<` should be used. For example, the following loop is wrong:

off-by-one error

```
for (int i = 0; i <= list.length; i++)
  System.out.print(list[i] + " ");
```

The `<=` should be replaced by `<`. Using a foreach loop can avoid the off-by-one error in this case.

Check Point

7.2.1 How do you declare an array reference variable and how do you create an array?

7.2.2 When is the memory allocated for an array?

7.2.3 Indicate true or false for the following statements:

a. Every element in an array has the same type.
b. The array size is fixed after an array reference variable is declared.
c. The array size is fixed after it is created.
d. The elements in an array must be of a primitive data type.

7.2.4 What is the output of the following code?

```
int x = 30;
int[] numbers = new int[x];
x = 60;
System.out.println("x is " + x);
System.out.println("The size of numbers is " + numbers.length);
```

7.2.5 How do you access elements in an array?

7.2.6 What is the array index type? What is the lowest index? What is the representation of the third element in an array named `a`?

7.2.7 Which of the following statements are valid?

a. `int i = new int(30);`
b. `double d[] = new double[30];`
c. `char[] r = new char(1..30);`
d. `int i[] = (3, 4, 3, 2);`
e. `float f[] = {2.3, 4.5, 6.6};`
f. `char[] c = new char();`

7.2.8 Write statements to do the following:

a. Create an array to hold `10` double values.
b. Assign the value `5.5` to the last element in the array.
c. Display the sum of the first two elements.
d. Write a loop that computes the sum of all elements in the array.
e. Write a loop that finds the minimum element in the array.
f. Randomly generate an index and display the element of this index in the array.
g. Use an array initializer to create another array with the initial values `3.5`, `5.5`, `4.52`, and `5.6`.

7.2.9 What happens when your program attempts to access an array element with an invalid index?

7.2.10 Identify and fix the errors in the following code:

```
public class Test {
  public static void main(String[] args) {
    double[100] r;

    for (int i = 0; i < r.length(); i++);
      r(i) = Math.random * 100;
  }
}
```

7.2.11 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    int list[] = {1, 2, 3, 4, 5, 6};
    for (int i = 1; i < list.length; i++)
      list[i] = list[i - 1];

    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }
}
```

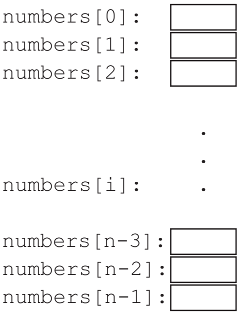
7.3 Case Study: Analyzing Numbers

The problem is to write a program that finds the number of items above the average of all items.

Key Point

Now you can write a program using arrays to solve the problem proposed at the beginning of this chapter. The problem is to read 100 numbers, get the average of these numbers, and find the number of the items greater than the average. To be flexible for handling any number of inputs, we will let the user enter the number of inputs, rather than fixing it to 100. Listing 7.1 gives a solution.

Listing 7.1 `AnalyzeNumbers.java`



create array

store number in array

```
 1 public class AnalyzeNumbers {
 2   public static void main(String[] args) {
 3     java.util.Scanner input = new java.util.Scanner(System.in);
 4     System.out.print("Enter the number of items: ");
 5     int n = input.nextInt();
 6     double[] numbers = new double[n];
 7     double sum = 0;
 8
 9     System.out.print("Enter the numbers: ");
10     for (int i = 0; i < n; i++) {
11       numbers[i] = input.nextDouble();
12       sum += numbers[i];
13     }
14
15     double average = sum / n;
16
17     int count = 0; // The number of elements above average
18     for (int i = 0; i < n; i++)
19       if (numbers[i] > average)
20         count++;
21
22     System.out.println("Average is " + average);
23     System.out.println("Number of elements above the average is "
24       + count);
25   }
26 }
```

get average

above average?

```
Enter the number of items: 10 ↵Enter
Enter the numbers: 3.4 5 6 1 6.5 7.8 3.5 8.5 6.3 9.5 ↵Enter
Average is 5.75
Number of elements above the average is 6
```

The program prompts the user to enter the array size (line 5) and creates an array with the specified size (line 6). The program reads the input, stores numbers into the array (line 11), adds each number to **`sum`** in line 12, and obtains the average (line 15). It then compares each number in the array with the average to count the number of values above the average (lines 7–20).

7.4 Case Study: Deck of Cards

Key Point

The problem is to create a program that will randomly select four cards from a deck of cards.

Say you want to write a program that will pick four cards at random from a deck of **52** cards. All the cards can be represented using an array named `deck`, filled with initial values **0–51**, as follows:

VideoNote
Deck of cards

```
int[] deck = new int[52];

// Initialize cards
for (int i = 0; i < deck.length; i++)
  deck[i] = i;
```

Card numbers **0–12**, **13–25**, **26–38**, and **39–51** represent 13 Spades, 13 Hearts, 13 Diamonds, and 13 Clubs, respectively, as shown in Figure 7.2. `cardNumber / 13` determines the suit of the card, and `cardNumber % 13` determines the rank of the card, as shown in Figure 7.3. After shuffling the array `deck`, pick the first four cards from `deck`. The program displays the cards from these four card numbers.

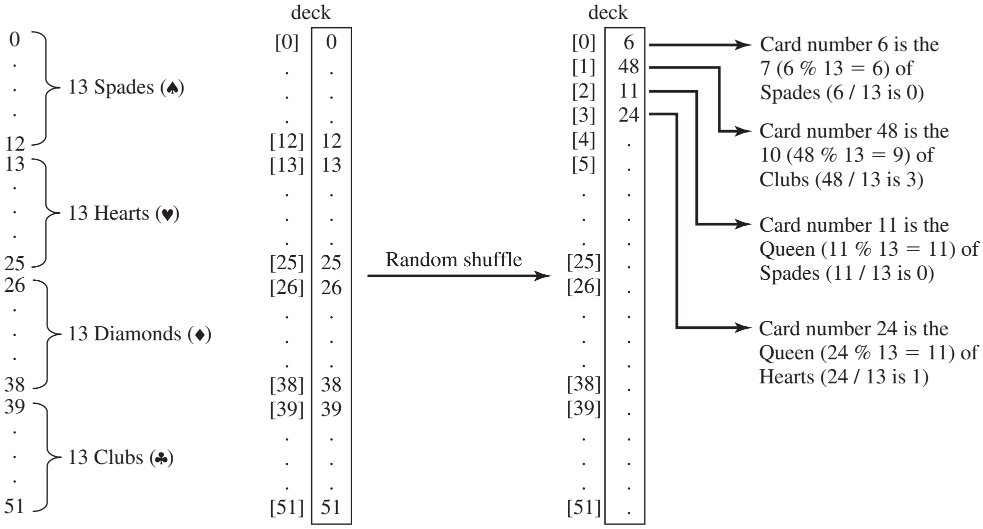

0 . . . 12 13 Spades (♠)
13 . . . 25 13 Hearts (♥)
26 . . . 38 13 Diamonds (♦)
39 . . . 51 13 Clubs (♣)
deck
[0] 0
[12] 12
[13] 13
[25] 25
[26] 26
[38] 38
[39] 39
[51] 51
Random shuffle
deck
[0] 6
[1] 48
[2] 11
[3] 24
[4]
[5]
[25]
[26]
[38]
[39]
[51]
Card number 6 is the 7 (6 % 13 = 6) of Spades (6 / 13 is 0)
Card number 48 is the 10 (48 % 13 = 9) of Clubs (48 / 13 is 3)
Card number 11 is the Queen (11 % 13 = 11) of Spades (11 / 13 is 0)
Card number 24 is the Queen (24 % 13 = 11) of Hearts (24 / 13 is 1)

FIGURE 7.2 52 cards are stored in an array named `deck`.

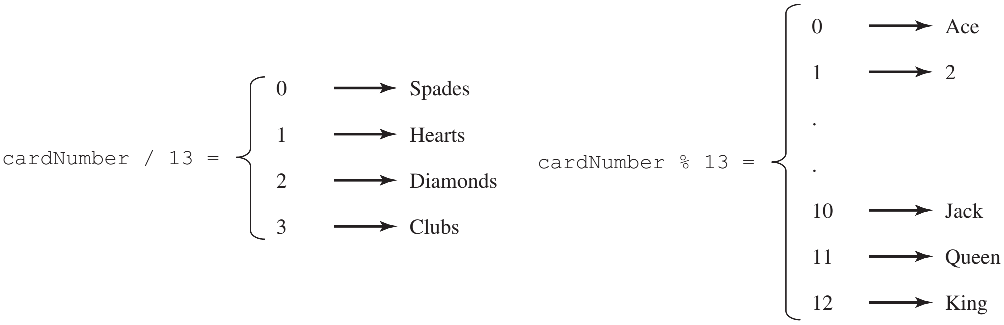

cardNumber / 13 =
0 → Spades
1 → Hearts
2 → Diamonds
3 → Clubs
cardNumber % 13 =
0 → Ace
1 → 2
.
.
10 → Jack
11 → Queen
12 → King

FIGURE 7.3 `cardNumber` identifies a card's suit and rank number.

Listing 7.2 gives the solution to the problem.

LISTING 7.2 DeckOfCards.java

```
 1  public class DeckOfCards {
 2    public static void main(String[] args) {
 3      int[] deck = new int[52];
 4      String[] suits = {"Spades", "Hearts", "Diamonds", "Clubs"};
 5      String[] ranks = {"Ace", "2", "3", "4", "5", "6", "7", "8", "9",
 6        "10", "Jack", "Queen", "King"};
 7
 8      // Initialize the cards
 9      for (int i = 0; i < deck.length; i++)
10        deck[i] = i;
11
12      // Shuffle the cards
13      for (int i = 0; i < deck.length; i++) {
14        // Generate an index randomly
15        int index = (int)(Math.random() * deck.length);
16        int temp = deck[i];
17        deck[i] = deck[index];
18        deck[index] = temp;
19      }
20
21      // Display the first four cards
22      for (int i = 0; i < 4; i++) {
23        String suit = suits[deck[i] / 13];
24        String rank = ranks[deck[i] % 13];
25        System.out.println("Card number " + deck[i] + ": "
26          + rank + " of " + suit);
27      }
28    }
29  }
```

create array `deck`
array of strings
initialize deck
shuffle deck
suit of a card
rank of a card

```
Card number 6: 7 of Spades
Card number 48: 10 of Clubs
Card number 11: Queen of Spades
Card number 24: Queen of Hearts
```

The program creates an array **`suits`** for four suits (line 4) and an array **`ranks`** for 13 cards in a suit (lines 5 and 6). Each element in these arrays is a string.

The program initializes **`deck`** with values **`0`**–**`51`** in lines 9 and 10. The **`deck`** value **`0`** represents the Ace of Spades, **`1`** represents the card 2 of Spades, **`13`** represents the Ace of Hearts, and **`14`** represents the 2 of Hearts.

Lines 13–19 randomly shuffle the deck. After a deck is shuffled, **`deck[i]`** contains an arbitrary value. **`deck[i] / 13`** is **`0`**, **`1`**, **`2`**, or **`3`**, which determines the suit (line 23). **`deck[i] % 13`** is a value between **`0`** and **`12`**, which determines the rank (line 24). If the **`suits`** array is not defined, you would have to determine the suit using a lengthy multiway **`if-else`** statement as follows:

```
if (deck[i] / 13 == 0)
  System.out.print("suit is Spades");
else if (deck[i] / 13 == 1)
  System.out.print("suit is Hearts");
else if (deck[i] / 13 == 2)
  System.out.print("suit is Diamonds");
else
  System.out.print("suit is Clubs");
```

With `suits = {"Spades", "Hearts", "Diamonds", "Clubs"}` created in an array, `suits[deck[i] / 13]` gives the suit for `deck[i]`. Using arrays greatly simplifies the solution for this program.

Check Point

7.4.1 Will the program pick four random cards if you replace lines 22–27 in Listing 7.2, DeckOfCards.java, with the following code?

```
for (int i = 0; i < 4; i++) {
  int cardNumber = (int)(Math.random() * deck.length);
  String suit = suits[cardNumber / 13];
  String rank = ranks[cardNumber % 13];
  System.out.println("Card number " + cardNumber + ": "
    + rank + " of " + suit);
}
```

7.5 Copying Arrays

Key Point

To copy the contents of one array into another, you have to copy the array's individual elements into the other array.

Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (`=`), as follows:

```
list2 = list1;
```

copy reference

garbage collection

However, this statement does not copy the contents of the array referenced by `list1` to `list2`, but instead merely copies the reference value from `list1` to `list2`. After this statement, `list1` and `list2` reference the same array, as shown in Figure 7.4. The array previously referenced by `list2` is no longer referenced; it becomes garbage, which will be automatically collected by the Java Virtual Machine. This process is called *garbage collection*.

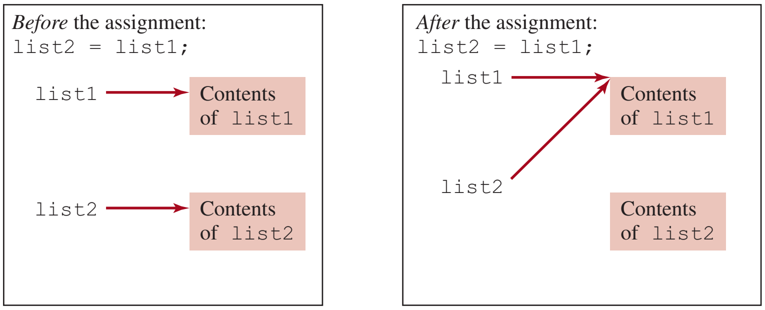


FIGURE 7.4 Before the assignment statement, `list1` and `list2` point to separate memory locations. After the assignment, the reference of the `list1` array is passed to `list2`.

In Java, you can use assignment statements to copy primitive data type variables, but not arrays. Assigning one array variable to another array variable actually copies one reference to another and makes both variables point to the same memory location.

There are three ways to copy arrays:

1. Use a loop to copy individual elements one by one.
2. Use the static `arraycopy` method in the `System` class.
3. Use the `clone` method to copy arrays; this will be introduced in Chapter 13, Abstract Classes and Interfaces.

You can write a loop to copy every element from the source array to the corresponding element in the target array. The following code, for instance, copies `sourceArray` to `targetArray` using a `for` loop:

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray.length];
for (int i = 0; i < sourceArray.length; i++) {
  targetArray[i] = sourceArray[i];
}
```

`arraycopy` method

Another approach is to use the `arraycopy` method in the `java.lang.System` class to copy arrays instead of using a loop. The syntax for `arraycopy` is:

```
arraycopy(sourceArray, srcPos, targetArray, tarPos, length);
```

The parameters `srcPos` and `tarPos` indicate the starting positions in `sourceArray` and `targetArray`, respectively. The number of elements copied from `sourceArray` to `targetArray` is indicated by `length`. For example, you can rewrite the loop using the following statement:

```
System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);
```

The `arraycopy` method does not allocate memory space for the target array. The target array must have already been created with its memory space allocated. After the copying takes place, `targetArray` and `sourceArray` have the same content but independent memory locations.

Note

The `arraycopy` method violates the Java naming convention. By convention, this method should be named `arrayCopy` (i.e., with an uppercase C).

Check Point

7.5.1 Use the `arraycopy` method to copy the following array to a target array `t`:

```
int[] source = {3, 4, 5};
```

7.5.2 Once an array is created, its size cannot be changed. Does the following code resize the array?

```
int[] myList;
myList = new int[10];
// Sometime later you want to assign a new array to myList
myList = new int[20];
```

7.6 Passing Arrays to Methods

Key Point

When passing an array to a method, the reference of the array is passed to the method.

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an `int` array:

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

You can invoke it by passing an array. For example, the following statement invokes the `printArray` method to display 3, 1, 2, 6, 4, and 2.

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Note

The preceding statement creates an array using the following syntax:

```
new elementType[]{value0, value1, ..., valuek};
```

anonymous array

There is no explicit reference variable for the array. Such array is called an *anonymous array*.

pass-by-value

Java uses *pass-by-value* to pass arguments to a method. There are important differences between passing the values of variables of primitive data types and passing arrays.

- For an argument of a primitive type, the argument's value is passed.
- For an argument of an array type, the value of the argument is a reference to an array; this reference value is passed to the method. Semantically, it can be best described as *pass-by-sharing*, that is, the array in the method is the same as the array being passed. Thus, if you change the array in the method, you will see the change outside the method.

pass-by-sharing

Take the following code, for example:

```
public class TestArrayArguments {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    m(x, y); // Invoke m with arguments x and y

    System.out.println("x is " + x);
    System.out.println("y[0] is " + y[0]);
  }

  public static void m(int number, int[] numbers) {
    number = 1001; // Assign a new value to number
    numbers[0] = 5555; // Assign a new value to numbers[0]
  }
}
```

```
x is 1
y[0] is 5555
```

You may wonder why after `m` is invoked, `x` remains `1`, but `y[0]` becomes `5555`. This is because `y` and `numbers`, although they are independent variables, reference the same array, as illustrated in Figure 7.5. When `m(x, y)` is invoked, the values of `x` and `y` are passed to `number` and `numbers`. Since `y` contains the reference value to the array, `numbers` now contains the same reference value to the same array.

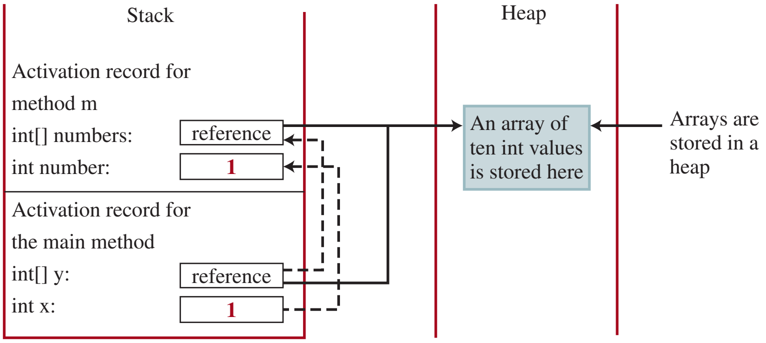

Stack
Heap
Activation record for method m
int[] numbers:
reference
int number:
1
Activation record for the main method
int[] y:
reference
int x:
1
An array of ten int values is stored here
Arrays are stored in a heap

FIGURE 7.5 The primitive type value in `x` is passed to `number`, and the reference value in `y` is passed to `numbers`.

Note

Arrays are objects in Java (objects are introduced in Chapter 9). The JVM stores the objects in an area of memory called the *heap*, which is used for dynamic memory allocation.

heap

Listing 7.3 gives another program that shows the difference between passing a primitive data type value and an array reference variable to a method.

The program contains two methods for swapping elements in an array. The first method, named `swap`, fails to swap two `int` arguments. The second method, named `swapFirstTwoInArray`, successfully swaps the first two elements in the array argument.

LISTING 7.3 TestPassArray.java

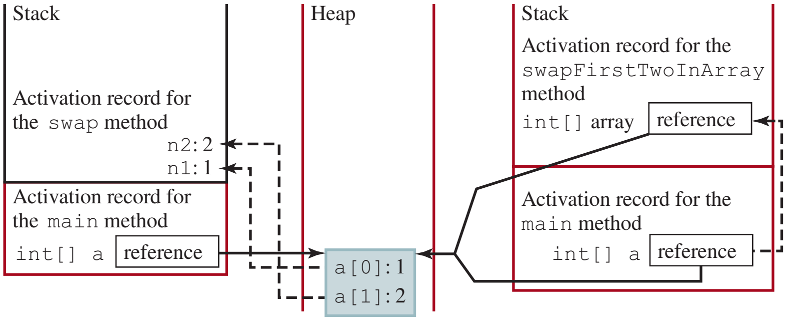
```java
public class TestPassArray {
  /** Main method */
  public static void main(String[] args) {
    int[] a = {1, 2};

    // Swap elements using the swap method
    System.out.println("Before invoking swap");
    System.out.println("array is {" + a[0] + ", " + a[1] + "}");
    swap(a[0], a[1]);
    System.out.println("After invoking swap");
    System.out.println("array is {" + a[0] + ", " + a[1] + "}");

    // Swap elements using the swapFirstTwoInArray method
    System.out.println("Before invoking swapFirstTwoInArray");
    System.out.println("array is {" + a[0] + ", " + a[1] + "}");
    swapFirstTwoInArray(a);
    System.out.println("After invoking swapFirstTwoInArray");
    System.out.println("array is {" + a[0] + ", " + a[1] + "}");
  }

  /** Swap two variables */
  public static void swap(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
  }

  /** Swap the first two elements in the array */
  public static void swapFirstTwoInArray(int[] array) {
    int temp = array[0];
    array[0] = array[1];
    array[1] = temp;
  }
}
```

false swap

swap array elements

```
Before invoking swap
array is {1, 2}
After invoking swap
array is {1, 2}
Before invoking swapFirstTwoInArray
array is {1, 2}
After invoking swapFirstTwoInArray
array is {2, 1}
```

As shown in Figure 7.6, the two elements are not swapped using the `swap` method. However, they are swapped using the `swapFirstTwoInArray` method. Since the parameters in the `swap` method are primitive type, the values of `a[0]` and `a[1]` are passed to `n1` and `n2` inside the method when invoking `swap(a[0], a[1])`. The memory locations for `n1` and `n2` are independent of the ones for `a[0]` and `a[1]`. The contents of the array are not affected by this call.


Stack
Activation record for the swap method
n2: 2
n1: 1
Activation record for the main method
int[] a
reference
Heap
a[0]: 1
a[1]: 2
Stack
Activation record for the swapFirstTwoInArray method
int[] array
reference
Activation record for the main method
int[] a
reference

Invoke `swap(int n1, int n2)`. The primitive type values in `a[0]` and `a[1]` are passed to the `swap` method.

The arrays are stored in a heap.

Invoke `swapFirstTwoInArray(int[] array)`. The reference value in `a` is passed to the `swapFirstTwoInArray` method.

FIGURE 7.6 When passing an array to a method, the reference of the array is passed to the method.

The parameter in the `swapFirstTwoInArray` method is an array. As shown in Figure 7.6, the reference of the array is passed to the method. Thus, the variables `a` (outside the method) and `array` (inside the method) both refer to the same array in the same memory location. Therefore, swapping `array[0]` with `array[1]` inside the method `swapFirstTwoInArray` is the same as swapping `a[0]` with `a[1]` outside of the method.

Check Point

7.6.1 True or false? When an array is passed to a method, a new array is created and passed to the method.

7.7 Returning an Array from a Method

Key Point

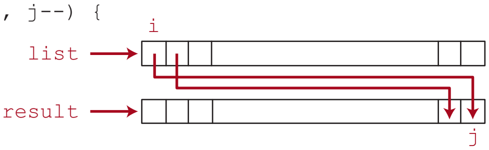
When a method returns an array, the reference of the array is returned.

You can pass arrays when invoking a method. A method may also return an array. For example, the following method returns an array that is the reversal of another array.

create array

return array

```
 1 public static int[] reverse(int[] list) {
 2   int[] result = new int[list.length];
 3
 4   for (int i = 0, j = result.length - 1;
 5        i < list.length; i++, j--) {
 6     result[j] = list[i];
 7   }
 8
 9   return result;
10 }
```


i
list
result
j

Line 2 creates a new array `result`. Lines 4–7 copy elements from array `list` to array `result`. Line 9 returns the array. For example, the following statement returns a new array `list2` with elements `6`, `5`, `4`, `3`, `2`, `1`:

```
int[] list1 = {1, 2, 3, 4, 5, 6};
int[] list2 = reverse(list1);
```

Check Point

7.7.1 Suppose the following code is written to reverse the contents in an array, explain why it is wrong. How do you fix it?

```
int[] list = {1, 2, 3, 5, 4};

for (int i = 0, j = list.length - 1; i < list.length; i++, j--) {
  // Swap list[i] with list[j]
  int temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

7.8 Case Study: Counting the Occurrences of Each Letter

Key Point

This section presents a program to count the occurrences of each letter in an array of characters.

The program given in Listing 7.4 does the following:

1. Generates `100` lowercase letters randomly and assigns them to an array of characters, as shown in Figure 7.7a. You can obtain a random letter by using the `getRandomLowerCaseLetter()` method in the `RandomCharacter` class in Listing 6.10.
2. Count the occurrences of each letter in the array. To do so, create an array, say `counts`, of `26 int` values, each of which counts the occurrences of a letter, as shown in Figure 7.7b. That is, `counts[0]` counts the number of `a`'s, `counts[1]` counts the number of `b`'s, and so on.

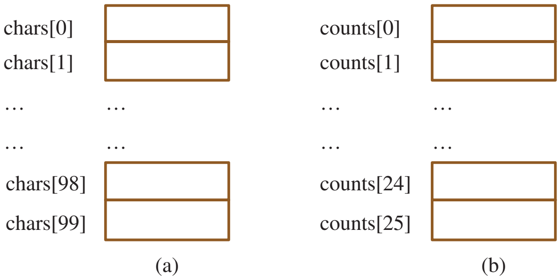


FIGURE 7.7 The `chars` array stores `100` characters, and the `counts` array stores `26` counts, each of which counts the occurrences of a letter.

LISTING 7.4 `CountLettersInArray.java`

```
public class CountLettersInArray {
  /** Main method */
  public static void main(String[] args) {
    // Declare and create an array
    char[] chars = createArray();

    // Display the array
    System.out.println("The lowercase letters are:");
    displayArray(chars);

```

create array

pass array

return array

pass array

increase count

```
    // Count the occurrences of each letter
    int[] counts = countLetters(chars);

    // Display counts
    System.out.println();
    System.out.println("The occurrences of each letter are:");
    displayCounts(counts);
  }

  /** Create an array of characters */
  public static char[] createArray() {
    // Declare an array of characters and create it
    char[] chars = new char[100];

    // Create lowercase letters randomly and assign
    // them to the array
    for (int i = 0; i < chars.length; i++)
      chars[i] = RandomCharacter.getRandomLowerCaseLetter();

    // Return the array
    return chars;
  }

  /** Display the array of characters */
  public static void displayArray(char[] chars) {
    // Display the characters in the array 20 on each line
    for (int i = 0; i < chars.length; i++) {
      if ((i + 1) % 20 == 0)
        System.out.println(chars[i]);
      else
        System.out.print(chars[i] + " ");
    }
  }

  /** Count the occurrences of each letter */
  public static int[] countLetters(char[] chars) {
    // Declare and create an array of 26 int
    int[] counts = new int[26];

    // For each lowercase letter in the array, count it
    for (int i = 0; i < chars.length; i++)
      counts[chars[i] - 'a']++;

    return counts;
  }

  /** Display counts */
  public static void displayCounts(int[] counts) {
    for (int i = 0; i < counts.length; i++) {
      if ((i + 1) % 10 == 0)
        System.out.println(counts[i] + " " + (char)(i + 'a'));
      else
        System.out.print(counts[i] + " " + (char)(i + 'a') + " ");
    }
  }
}
```

```
The lowercase letters are:
e y l s r i b k j v j h a b z n w b t v
s c c k r d w a m p w v u n q a m p l o
a z g d e g f i n d x m z o u l o z j v
h w i w n t g x w c d o t x h y v z y z
q e a m f w p g u q t r e n n w f c r f

The occurrences of each letter are:
5 a 3 b 4 c 4 d 4 e 4 f 4 g 3 h 3 i 3 j
2 k 3 l 4 m 6 n 4 o 3 p 3 q 4 r 2 s 4 t
3 u 5 v 8 w 3 x 3 y 6 z
```

The `createArray` method (lines 21–32) generates an array of `100` random lowercase letters. Line 5 invokes the method and assigns the array to `chars`. What would be wrong if you rewrote the code as follows?

```java
char[] chars = new char[100];
chars = createArray();
```

You would be creating two arrays. The first line would create an array by using `new char[100]`. The second line would create an array by invoking `createArray()` and assign the reference of the array to `chars`. The array created in the first line would be garbage because it is no longer referenced, and as mentioned earlier, Java automatically collects garbage behind the scenes. Your program would compile and run correctly, but it would create an array unnecessarily.

Invoking `getRandomLowerCaseLetter()` (line 28) returns a random lowercase letter. This method is defined in the `RandomCharacter` class in Listing 6.10.

The `countLetters` method (lines 46–55) returns an array of `26` `int` values, each of which stores the number of occurrences of a letter. The method processes each letter in the array and increases its count by one. A brute-force approach to count the occurrences of each letter might be as follows:

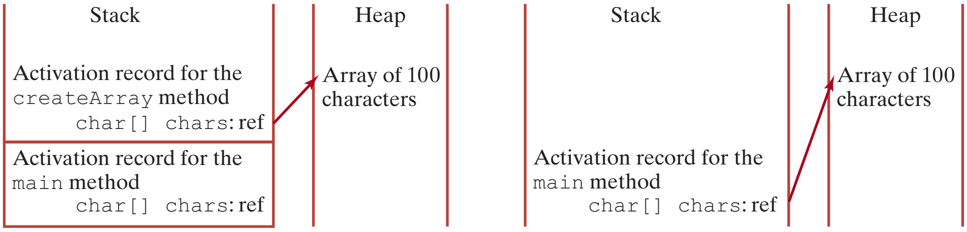
```java
for (int i = 0; i < chars.length; i++)
  if (chars[i] == 'a')
    counts[0]++;
  else if (chars[i] == 'b')
    counts[1]++;
  ...
```

However, a better solution is given in lines 51 and 52.

```java
    for (int i = 0; i < chars.length; i++)
      counts[chars[i] - 'a']++;
```

If the letter (`chars[i]`) is `a`, the corresponding count is `counts['a' - 'a']` (i.e., `counts[0]`). If the letter is `b`, the corresponding count is `counts['b' - 'a']` (i.e., `counts[1]`), since the Unicode of `b` is one more than that of `a`. If the letter is `z`, the corresponding count is `counts['z' - 'a']` (i.e., `counts[25]`), since the Unicode of `z` is `25` more than that of `a`.

Figure 7.8 shows the call stack and heap *during* and *after* executing `createArray`. See CheckPoint Question 7.8.3 to show the call stack and heap for other methods in the program.


Stack
Heap
Activation record for the createArray method
char[] chars: ref
Activation record for the main method
char[] chars: ref
Array of 100 characters
Stack
Heap
Activation record for the main method
char[] chars: ref
Array of 100 characters

(a) Executing `createArray` in line 5

(b) After exiting `createArray` in line 5

Figure 7.8 (a) An array of 100 characters is created when executing **`createArray`**. (b) This array is returned and assigned to the variable **`chars`** in the **`main`** method.

Check Point

7.8.1 Show the output of the following two programs:

```
public class Test {
  public static void main(String[] args) {
    int number = 0;
    int[] numbers = new int[1];

    m(number, numbers);

    System.out.println("number is " + number
      + " and numbers[0] is " + numbers[0]);
  }

  public static void m(int x, int[] y) {
    x = 3;
    y[0] = 3;
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    int[] list = {1, 2, 3, 4, 5};
    reverse(list);
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }

  public static void reverse(int[] list) {
    int[] newList = new int[list.length];

    for (int i = 0; i < list.length; i++)
      newList[i] = list[list.length - 1 - i];

    list = newList;
  }
}
```

(b)

7.8.2 Where are the arrays stored during execution? Show the contents of the stack and heap during and after executing **`displayArray`**, **`countLetters`**, and **`displayCounts`** in Listing 7.4.

7.9 Variable-Length Argument Lists

Key Point

A variable number of arguments of the same type can be passed to a method and treated as an array.

You can pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

```
typeName... parameterName
```

In the method declaration, you specify the type followed by an ellipsis (`...`). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Java treats a variable-length parameter as an array. You can pass an array or a variable number of arguments to a variable-length parameter. When invoking a method with a variable number of arguments, Java creates an array and passes the arguments to it. Listing 7.5 presents a method that prints the maximum value in a list of an unspecified number of values.

LISTING 7.5 `VarArgsDemo.java`

```
 1  public class VarArgsDemo {
 2    public static void main(String[] args) {
 3      printMax(34, 3, 3, 2, 56.5);
 4      printMax(new double[]{1, 2, 3});
 5    }
 6
 7    public static void printMax(double... numbers) {
 8      if (numbers.length == 0) {
 9        System.out.println("No argument passed");
10        return;
11      }
12
13      double result = numbers[0];
14
15      for (int i = 1; i < numbers.length; i++)
16        if (numbers[i] > result)
17          result = numbers[i];
18
19      System.out.println("The max value is " + result);
20    }
21  }
```

pass variable-length arg list

pass an array arg

a variable-length arg parameter

Line 3 invokes the `printMax` method with a variable-length argument list passed to the array `numbers`. If no arguments are passed, the length of the array is `0` (line 8).

Line 4 invokes the `printMax` method with an array.

Check Point

7.9.1 What is wrong with each of the following methods headers?

```
a. public static void print(String... strings, double... numbers)
b. public static void print(double... numbers, String name)
c. public static double... print(double d1, double d2)
```

7.9.2 Can you invoke the `printMax` method in Listing 7.5 using the following statements?

```
a. printMax(1, 2, 2, 1, 4);
b. printMax(new double[]{1, 2, 3});
c. printMax(new int[]{1, 2, 3});
```

7.10 Searching Arrays

Key Point

If an array is sorted, binary search is more efficient than linear search for finding an element in the array.

Searching is the process of looking for a specific element in an array—for example, discovering whether a certain score is included in a list of scores. Searching is a common task in computer programming. Many algorithms and data structures are devoted to searching. This section discusses two commonly used approaches, *linear search* and *binary search*.

linear search

binary search

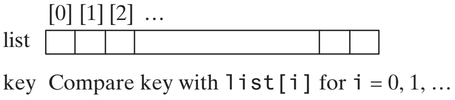
7.10.1 The Linear Search Approach

The linear search approach compares the key element `key` sequentially with each element in the array. It continues to do so until the key matches an element in the array, or the array is exhausted without a match being found. If a match is made, the linear search returns the index of the element in the array that matches the key. If no match is found, the search returns `-1`. The `linearSearch` method in Listing 7.6 gives the solution.

linear search animation on Companion Website

LISTING 7.6 `LinearSearch.java`

```java
public class LinearSearch {
  /** The method for finding a key in the list */
  public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++) {
      if (key == list[i])
        return i;
    }
    return -1;
  }
}
```



To better understand this method, trace it with the following statements:

```java
int[] list = {1, 4, 4, 2, 5, -3, 6, 2};
int i = linearSearch(list, 4);  // Returns 1
int j = linearSearch(list, -4); // Returns -1
int k = linearSearch(list, -3); // Returns 5
```

The linear search method compares the key with each element in the array. The elements can be in any order. On average, the algorithm will have to examine half of the elements in an array before finding the key, if it exists. Since the execution time of a linear search increases linearly as the number of array elements increases, linear search is inefficient for a large array.

7.10.2 The Binary Search Approach

Binary search is the other common search approach for a list of values. For binary search to work, the elements in the array must already be ordered. Assume that the array is in ascending order. The binary search first compares the key with the element in the middle of the array. Consider the following three cases:

1. If the key is less than the middle element, you need to continue to search for the key only in the first half of the array.
2. If the key is equal to the middle element, the search ends with a match.
3. If the key is greater than the middle element, you need to continue to search for the key only in the second half of the array.

binary search animation on Companion Website

Clearly, the binary search method eliminates at least half of the array after each comparison. Suppose the array has *n* elements. For convenience, let `n` be a power of `2`. After the first comparison, `n/2` elements are left for further search; after the second comparison, `(n/2)/2` elements are left. After the `k`th comparison, $n/2^k$ elements are left for further search. When $k = \log_2 n$, only one element is left in the array, and you need only one more comparison. Therefore, in the worst case when using the binary search approach, you need $\log_2 n+1$ comparisons to find an element in the sorted array. In the worst case for a list of `1024` (2^{10}) elements, binary search requires only `11` comparisons, whereas a linear search requires `1024` comparisons in the worst case.

The portion of the array being searched shrinks by half after each comparison. Let `low` and `high` denote, respectively, the first index and last index of the array that is currently being searched. Initially, `low` is `0` and `high` is `list.length - 1`. Let `mid` denote the index of the middle element, so `mid` is `(low + high)/2`. Figure 7.9 shows how to find key `11` in the list {`2`, `4`, `7`, `10`, `11`, `45`, `50`, `59`, `60`, `66`, `69`, `70`, `79`} using binary search.

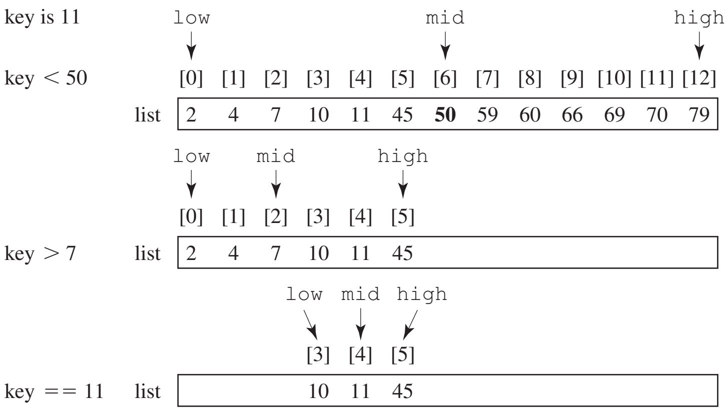


FIGURE 7.9 Binary search eliminates half of the list from further consideration after each comparison.

You now know how the binary search works. The next task is to implement it in Java. Don't rush to give a complete implementation. Implement it incrementally, one step at a time. You may start with the first iteration of the search, as shown in Figure 7.10a. It compares the key with the middle element in the list whose `low` index is `0` and `high` index is `list.length - 1`. If `key < list[mid]`, set the `high` index to `mid - 1`; if `key == list[mid]`, a match is found and return `mid`; if `key > list[mid]`, set the `low` index to `mid + 1`.

Next, consider implementing the method to perform the search repeatedly by adding a loop, as shown in Figure 7.10b. The search ends if the key is found, or if the key is not found when `low > high`.

why not −1?

```java
public static int binarySearch(
    int[] list, int key) {
  int low = 0;
  int high = list.length - 1;


    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;




}
```

(a) Version 1

```java
public static int binarySearch(
    int[] list, int key) {
  int low = 0;
  int high = list.length - 1;

  while (high >= low) {
    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;
  }

  return -1; // Not found
}
```

(b) Version 2

FIGURE 7.10 Binary search is implemented incrementally.

When the key is not found, `low` is the insertion point where a key would be inserted to maintain the order of the list. It is more useful to return the insertion point than `-1`. The method must return a negative value to indicate that the key is not in the list. Can it simply return `-low`? No. If the key is less than `list[0]`, `low` would be `0`. `-0` is `0`. This would indicate the key matches `list[0]`. A good choice is to let the method return `-low - 1` if the key is not in the list. Returning `-low - 1` indicates not only that the key is not in the list, but also where the key would be inserted.

The complete program is given in Listing 7.7.

Listing 7.7 `BinarySearch.java`

```java
1  public class BinarySearch {
2    /** Use binary search to find the key in the list */
3    public static int binarySearch(int[] list, int key) {
4      int low = 0;
5      int high = list.length - 1;
6
7      while (high >= low) {
8        int mid = (low + high) / 2;
9        if (key < list[mid])
10         high = mid - 1;
11       else if (key == list[mid])
12         return mid;
13       else
14         low = mid + 1;
15     }
16
17     return -low - 1; // Now high < low, key not found
18   }
19 }
```

first half

second half

The binary search returns the index of the search key if it is contained in the list (line 12). Otherwise, it returns `-low - 1` (line 17).

What would happen if we replaced `(high >= low)` in line 7 with (`high > low`)? The search would miss a possible matching element. Consider a list with just one element. The search would miss the element.

Does the method still work if there are duplicate elements in the list? Yes, as long as the elements are sorted in increasing order. The method returns the index of one of the matching elements if the element is in the list.

precondition

postcondition

The precondition for the binary search method is that the list must be sorted in increasing order. The postcondition is that the method returns the index of the element that matches the key if the key is in the list or a negative integer `k` such that `-k - 1` is the position for inserting the key. Precondition and postcondition are the terms often used to describe the properties of a method. Preconditions are the things that are true before the method is invoked, and postconditions are the things that are true after the method is returned:

To better understand this method, trace it with the following statements and identify `low` and `high` when the method returns.

```java
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
int i = BinarySearch.binarySearch(list, 2); // Returns 0
int j = BinarySearch.binarySearch(list, 11); // Returns 4
int k = BinarySearch.binarySearch(list, 12); // Returns -6
int l = BinarySearch.binarySearch(list, 1); // Returns -1
int m = BinarySearch.binarySearch(list, 3); // Returns -2
```

Here is the table that lists the `low` and `high` values when the method exits, and the value returned from invoking the method.

Method	Low	High	Value Returned
`binarySearch(list, 2)`	0	1	0 (mid)
`binarySearch(list, 11)`	3	5	4 (mid)
`binarySearch(list, 12)`	5	4	-6
`binarySearch(list, 1)`	0	-1	-1
`binarySearch(list, 3)`	1	0	-2

Note

Linear search is useful for finding an element in a small array or an unsorted array, but it is inefficient for large arrays. Binary search is more efficient, but it requires that the array be presorted.

binary search benefits

Check Point

7.10.1 If `high` is a very large integer such as the maximum `int` value `2147483647`, `(low + high) / 2` may cause overflow. How do you fix it to avoid overflow?

7.10.2 Use Figure 7.9 as an example to show how to apply the binary search approach to a search for key `10` and key `12` in list {`2`, `4`, `7`, `10`, `11`, `45`, `50`, `59`, `60`, `66`, `69`, `70`, `79`}.

7.10.3 If the binary search method returns `-4`, is the key in the list? Where should the key be inserted if you wish to insert the key into the list?

7.11 Sorting Arrays

Key Point

Sorting, like searching, is a common task in computer programming. Many different algorithms have been developed for sorting. This section introduces an intuitive sorting algorithm: selection sort.

VideoNote

Selection sort

Suppose you want to sort a list in ascending order. Selection sort finds the smallest number in the list and swaps it with the first element. It then finds the smallest number remaining and swaps it with the second element, and so on, until only a single number remains. Figure 7.11 shows how to sort the list {`2`, `9`, `5`, `4`, `8`, `1`, `6`} using selection sort.

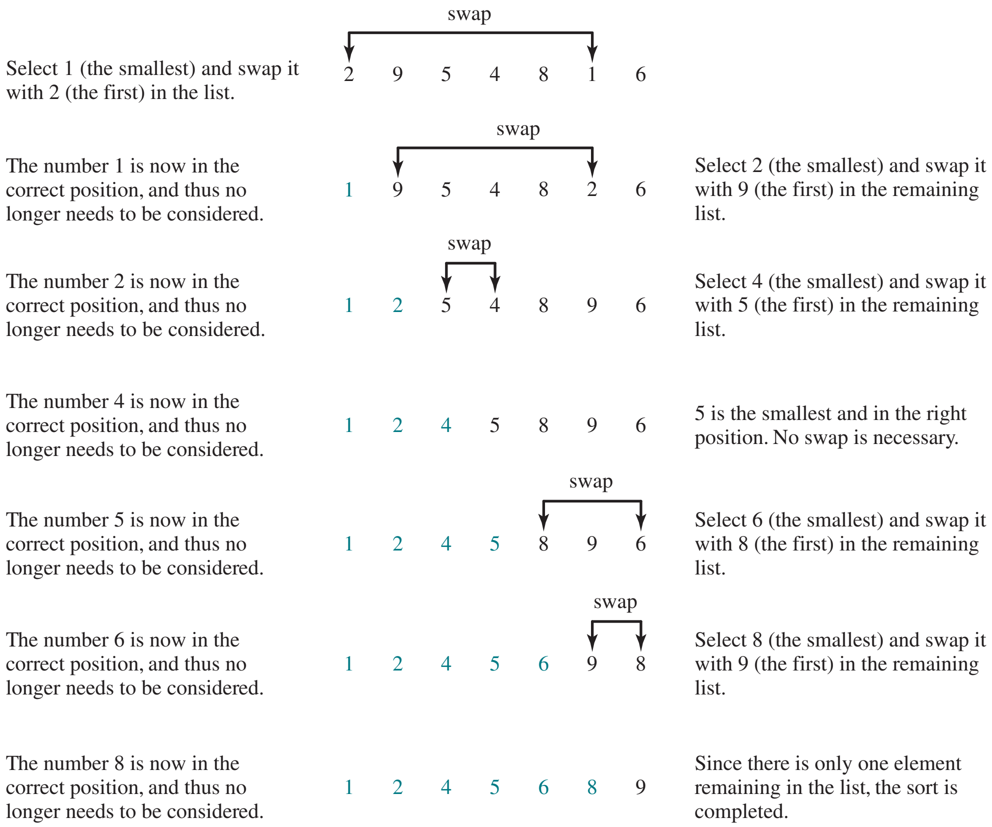

swap
Select 1 (the smallest) and swap it with 2 (the first) in the list.
2 9 5 4 8 1 6
swap
The number 1 is now in the correct position, and thus no longer needs to be considered.
1 9 5 4 8 2 6
Select 2 (the smallest) and swap it with 9 (the first) in the remaining list.
swap
The number 2 is now in the correct position, and thus no longer needs to be considered.
1 2 5 4 8 9 6
Select 4 (the smallest) and swap it with 5 (the first) in the remaining list.
The number 4 is now in the correct position, and thus no longer needs to be considered.
1 2 4 5 8 9 6
5 is the smallest and in the right position. No swap is necessary.
swap
The number 5 is now in the correct position, and thus no longer needs to be considered.
1 2 4 5 8 9 6
Select 6 (the smallest) and swap it with 8 (the first) in the remaining list.
swap
The number 6 is now in the correct position, and thus no longer needs to be considered.
1 2 4 5 6 9 8
Select 8 (the smallest) and swap it with 9 (the first) in the remaining list.
The number 8 is now in the correct position, and thus no longer needs to be considered.
1 2 4 5 6 8 9
Since there is only one element remaining in the list, the sort is completed.

FIGURE 7.11 Selection sort repeatedly selects the smallest number and swaps it with the first number in the list.

You know how the selection-sort approach works. The task now is to implement it in Java. Beginners find it difficult to develop a complete solution on the first attempt. Start by writing the code for the first iteration to find the smallest element in the list and swap it with the first element, then observe what would be different for the second iteration, the third, and so on. The insight this gives will enable you to write a loop that generalizes all the iterations.

selection sort animation on Companion Website

The solution can be described as follows:

```
for (int i = 0; i < list.length - 1; i++) {
  select the smallest element in list[i..list.length-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration applies on list[i+1..list.length-1]
}
```

Listing 7.8 implements the solution.

Listing 7.8 `SelectionSort.java`

```
 1 public class SelectionSort {
 2   /** The method for sorting the numbers */
 3   public static void selectionSort(double[] list) {
 4     for (int i = 0; i < list.length - 1; i++) {
 5       // Find the minimum in the list[i..list.length-1]
 6       double currentMin = list[i];
 7       int currentMinIndex = i;
 8
 9       for (int j = i + 1; j < list.length; j++) {
10         if (currentMin > list[j]) {
11           currentMin = list[j];
12           currentMinIndex = j;
13         }
14       }
15
16       // Swap list[i] with list[currentMinIndex] if necessary
17       if (currentMinIndex != i) {
18         list[currentMinIndex] = list[i];
19         list[i] = currentMin;
20       }
21     }
22   }
23 }
```

select

swap

The `selectionSort(double[] list)` method sorts any array of `double` elements. The method is implemented with a nested `for` loop. The outer loop (with the loop control variable `i` in line 4) is iterated in order to find the smallest element in the list, which ranges from `list[i]` to `list[list.length-1]`, and exchanges it with `list[i]`.

The variable `i` is initially `0`. After each iteration of the outer loop, `list[i]` is in the right place. Eventually, all the elements are put in the right place; therefore, the whole list is sorted.

To understand this method better, trace it with the following statements:

```
double[] list = {1, 9, 4.5, 6.6, 5.7, -4.5};
SelectionSort.selectionSort(list);
```

Check Point

7.11.1 Use Figure 7.11 as an example to show how to apply the selection-sort approach to sort {`3.4`, `5`, `3`, `3.5`, `2.2`, `1.9`, `2`}.

7.11.2 How do you modify the `selectionSort` method in Listing 7.8 to sort numbers in decreasing order?

Key Point

7.12 The Arrays Class

The `java.util.Arrays` *class contains useful methods for common array operations such as sorting and searching.*

The **`java.util.Arrays`** class contains various static methods for sorting and searching arrays, comparing arrays, filling array elements, and returning a string representation of the array. These methods are overloaded for all primitive types.

sort
parallelSort

You can use the **`sort`** or **`parallelSort`** method to sort a whole array or a partial array. For example, the following code sorts an array of numbers and an array of characters:

```
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers); // Sort the whole array
java.util.Arrays.parallelSort(numbers); // Sort the whole array

char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
java.util.Arrays.sort(chars, 1, 3); // Sort part of the array
java.util.Arrays.parallelSort(chars, 1, 3); // Sort part of the array
```

Invoking **`sort(numbers)`** sorts the whole array **`numbers`**. Invoking **`sort(chars, 1, 3)`** sorts a partial array from **`chars[1]`** to **`chars[3-1]`**. **`parallelSort`** is more efficient if your computer has multiple processors.

binarySearch

You can use the **`binarySearch`** method to search for a key in an array. The array must be presorted in increasing order. If the key is not in the array, the method returns **`-(insertionIndex + 1)`**. For example, the following code searches the keys in an array of integers and an array of characters:

```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
System.out.println("1. Index is " +
  java.util.Arrays.binarySearch(list, 11));
System.out.println("2. Index is " +
  java.util.Arrays.binarySearch(list, 12));

char[] chars = {'a', 'c', 'g', 'x', 'y', 'z'};
System.out.println("3. Index is " +
  java.util.Arrays.binarySearch(chars, 'a'));
System.out.println("4. Index is " +
  java.util.Arrays.binarySearch(chars, 't'));
```

The output of the preceding code is as follows:

```
Index is 4

Index is -6

Index is 0

Index is -4
```

equals

You can use the **`equals`** method to check whether two arrays are strictly equal. Two arrays are strictly equal if their corresponding elements are the same. In the following code, **`list1`** and **`list2`** are equal, but **`list2`** and **`list3`** are not.

```
int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 10};
int[] list3 = {4, 2, 7, 10};
System.out.println(java.util.Arrays.equals(list1, list2)); // true
System.out.println(java.util.Arrays.equals(list2, list3)); // false
```

toString
fill

You can use the **`fill`** method to fill in all or part of the array. For example, the following code fills **`list1`** with **`5`** and fills **`8`** into elements **`list2[1]`** through **`list2[5-1]`**.

```
int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 7, 7, 10};
java.util.Arrays.fill(list1, 5); // Fill 5 to the whole array
java.util.Arrays.fill(list2, 1, 5, 8); // Fill 8 to a partial array
```

You can also use the `toString` method to return a string that represents all elements in the array. This is a quick and simple way to display all elements in the array. For example, the following code:

```
int[] list = {2, 4, 7, 10};
System.out.println(java.util.Arrays.toString(list));
```

displays `[2, 4, 7, 10]`.

Check Point

7.12.1 What types of array can be sorted using the `java.util.Arrays.sort` method? Does this `sort` method create a new array?

7.12.2 To apply `java.util.Arrays.binarySearch(array, key)`, should the array be sorted in increasing order, in decreasing order, or neither?

7.12.3 Show the output of the following code:

```
int[] list1 = {2, 4, 7, 10};
java.util.Arrays.fill(list1, 7);
System.out.println(java.util.Arrays.toString(list1));

int[] list2 = {2, 4, 7, 10};
System.out.println(java.util.Arrays.toString(list2));
System.out.print(java.util.Arrays.equals(list1, list2));
```

7.13 Command-Line Arguments

Key Point

The `main` method can receive string arguments from the command line.

Perhaps you have already noticed the unusual header for the `main` method, which has the parameter `args` of the `String[]` type. It is clear that `args` is an array of strings. The `main` method is just like a regular method with a parameter. You can call a regular method by passing actual parameters. Can you pass arguments to `main`? Yes, of course you can. In the following examples, the `main` method in class `TestMain` is invoked by a method in `A`:

```
public class A {
  public static void main(String[] args) {
    String[] strings = {"New York",
      "Boston", "Atlanta"};
    TestMain.main(strings);
  }
}
```

```
public class TestMain {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

A `main` method is just like a regular method. Furthermore, you can pass arguments to a main method from the command line.

7.13.1 Passing Strings to the `main` Method

You can pass strings to a `main` method from the command line when you run the program. The following command line, for example, starts the program `TestMain` with three strings: `arg0`, `arg1`, and `arg2`:

```
java TestMain arg0 arg1 arg2
```

`arg0`, `arg1`, and `arg2` are strings, but they don't have to appear in double quotes on the command line. The strings are separated by a space. A string that contains a space must be enclosed in double quotes. Consider the following command line:

```
java TestMain "First num" alpha 53
```

It starts the program with three strings: `First num`, `alpha`, and `53`. Since `First num` is a string, it is enclosed in double quotes. Note `53` is actually treated as a string. You can use `"53"` instead of `53` in the command line.

When the `main` method is invoked, the Java interpreter creates an array to hold the command-line arguments and pass the array reference to `args`. For example, if you invoke a program with `n` arguments, the Java interpreter creates an array such as the one that follows:

```
args = new String[n];
```

The Java interpreter then passes `args` to invoke the `main` method.

Note
If you run the program with no strings passed, the array is created with `new String[0]`. In this case, the array is empty with length `0`. `args` references to this empty array. Therefore, `args` is not `null`, but `args.length` is `0`.

7.13.2 Case Study: Calculator

VideoNote
Command-line arguments

Suppose you are to develop a program that performs arithmetic operations on integers. The program receives an expression. The expression consists of an integer followed by an operator and another integer. For example, to add two integers, use this command:

```
java Calculator 2 + 3
```

The program will display the following output:

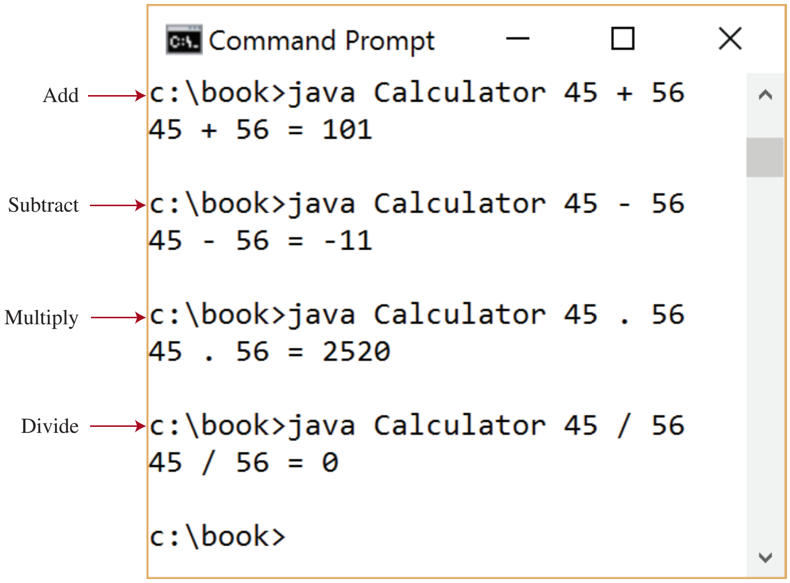
```
2 + 3 = 5
```

Figure 7.12 shows sample runs of the program.

The strings passed to the main program are stored in `args`, which is an array of strings. The first string is stored in `args[0]`, and `args.length` is the number of strings passed.

Here are the steps in the program:

1. Use `args.length` to determine whether the expression has been provided as three arguments in the command line. If not, terminate the program using `System.exit(1)`.
2. Perform a binary arithmetic operation on the operands `args[0]` and `args[2]` using the operator in `args[1]`.



```
c:\book>java Calculator 45 + 56
45 + 56 = 101

c:\book>java Calculator 45 - 56
45 - 56 = -11

c:\book>java Calculator 45 . 56
45 . 56 = 2520

c:\book>java Calculator 45 / 56
45 / 56 = 0

c:\book>
```

FIGURE 7.12 The program takes three arguments (`operand1 operator operand2`) from the command line and displays the expression and the result of the arithmetic operation.

The program is given in Listing 7.9.

LISTING 7.9 `Calculator.java`

check argument

check operator

```
 1  public class Calculator {
 2    /** Main method */
 3    public static void main(String[] args) {
 4      // Check number of strings passed
 5      if (args.length != 3) {
 6        System.out.println(
 7          "Usage: java Calculator operand1 operator operand2");
 8        System.exit(1);
 9      }
10
11      // The result of the operation
12      int result = 0;
13
14      // Determine the operator
15      switch (args[1].charAt(0)) {
16        case '+': result = Integer.parseInt(args[0]) +
17                           Integer.parseInt(args[2]);
18                  break;
19        case '-': result = Integer.parseInt(args[0]) -
20                           Integer.parseInt(args[2]);
21                  break;
22        case '.': result = Integer.parseInt(args[0]) *
23                           Integer.parseInt(args[2]);
24                  break;
25        case '/': result = Integer.parseInt(args[0]) /
26                           Integer.parseInt(args[2]);
27      }
28
29      // Display result
30      System.out.println(args[0] + ' ' + args[1] + ' ' + args[2]
31        + " = " + result);
32    }
33  }
```

`Integer.parseInt(args[0])` (line 16) converts a digital string into an integer. The string must consist of digits. If not, the program will terminate abnormally.

We used the `.` symbol for multiplication, not the common `*` symbol. The reason for this is the `*` symbol refers to all the files in the current directory when it is used on a command line. The following program displays all the files in the current directory when issuing the command `java Test *`:

```
public class Test {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
       System.out.println(args[i]);
  }
}
```

To circumvent this problem, we will have to use a different symbol for the multiplication operator.

Check Point

7.13.1 This book declares the `main` method as

```
public static void main(String[] args)
```

Can it be replaced by one of the following lines?

```
a. public static void main(String args[])
b. public static void main(String[] x)
```

```
c. public static void main(String x[])
d. static void main(String x[])
```

7.13.2 Show the output of the following program when invoked using

1. **java Test I have a dream**
2. **java Test "1 2 3"**
3. **java Test**

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Number of strings is " + args.length);
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

KEY TERMS

anonymous array 262
array 250
array initializer 252
binary search 269
garbage collection 260
index 250
indexed variable 252
linear search 269
off-by-one error 255
postcondition 272
precondition 272
selection sort 273

CHAPTER SUMMARY

1. A variable is declared as an *array* type using the syntax `elementType[] arrayRefVar` or `elementType arrayRefVar[]`. The style `elementType[] arrayRefVar` is preferred, although `elementType arrayRefVar[]` is legal.

2. Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. An array variable is not a primitive data type variable. An array variable contains a reference to an array.

3. You cannot assign elements to an array unless it has already been created. You can create an array by using the `new` operator with the following syntax: `new elementType[arraySize]`.

4. Each element in the array is represented using the syntax `arrayRefVar[index]`. An *index* must be an integer or an integer expression.

5. After an array is created, its size becomes permanent and can be obtained using `arrayRefVar.length`. Since the index of an array always begins with `0`, the last index is always `arrayRefVar.length – 1`. An out-of-bounds error will occur if you attempt to reference elements beyond the bounds of an array.

6. Programmers often mistakenly reference the first element in an array with index `1`, but it should be `0`. This is called the index *off-by-one error*.

7. When an array is created, its elements are assigned the default value of `0` for the numeric primitive data types, `\u0000` for char types, and `false` for `boolean` types.

8. Java has a shorthand notation, known as the *array initializer*, which combines declaring an array, creating an array, and initializing an array in one statement, using the syntax `elementType[] arrayRefVar = {value0, value1, . . . , valuek}`.

9. When you pass an array argument to a method, you are actually passing the reference of the array; that is, the called method can modify the elements in the caller's original array.

10. If an array is sorted, *binary search* is more efficient than *linear search* for finding an element in the array.

11. *Selection sort* finds the smallest number in the list and swaps it with the first element. It then finds the smallest number remaining and swaps it with the first element in the remaining list, and so on, until only a single number remains.

QUIZ

Answer the quiz for this chapter online at the Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

Sections 7.2–7.5

***7.1** (*Assign grades*) Write a program that reads student scores, gets the best score, and then assigns grades based on the following scheme:

Grade is A if score is $\geq$ best -10;
Grade is B if score is $\geq$ best -20;
Grade is C if score is $\geq$ best -30;
Grade is D if score is $\geq$ best -40;
Grade is F otherwise.

The program prompts the user to enter the total number of students, then prompts the user to enter all of the scores, and concludes by displaying the grades. Here is a sample run:

```
Enter the number of students: 4 ↵Enter
Enter 4 scores: 40 55 70 58 ↵Enter
Student 0 score is 40 and grade is C
Student 1 score is 55 and grade is B
Student 2 score is 70 and grade is A
Student 3 score is 58 and grade is B
```

7.2 (*Reverse the numbers entered*) Write a program that reads 10 integers then displays them in the reverse of the order in which they were read.

****7.3** (*Count occurrence of numbers*) Write a program that reads the integers between 1 and 100 and counts the occurrences of each. Assume the input ends with `0`. Here is a sample run of the program: Note that if a number occurs more than one time, the plural word "times" is used in the output. Numbers are displayed in increasing order.

```
Enter the integers between 1 and 100: 2 5 6 5 4 3 23 43 2 0 ↵Enter
2 occurs 2 times
3 occurs 1 time
4 occurs 1 time
5 occurs 2 times
6 occurs 1 time
23 occurs 1 time
43 occurs 1 time
```

7.4 (*Analyze scores*) Write a program that reads an unspecified number of scores and determines how many scores are above or equal to the average, and how many scores are below the average. Enter a negative number to signify the end of the input. Assume the maximum number of scores is 100.

****7.5** (*Print distinct numbers*) Write a program that reads in 10 numbers and displays the number of distinct numbers and the distinct numbers in their input order and separated by exactly one space (i.e., if a number appears multiple times, it is displayed only once). (*Hint*: Read a number and store it to an array if it is new. If the number is already in the array, ignore it.) After the input, the array contains the distinct numbers. Here is the sample run of the program:

```
Enter 10 numbers: 1 2 3 2 1 6 3 4 5 2 ↵Enter
The number of distinct numbers is 6
The distinct numbers are: 1 2 3 6 4 5
```

***7.6** (*Revise Listing 5.15, PrimeNumber.java*) Listing 5.15 determines whether a number `n` is prime by checking whether `2`, `3`, `4`, `5`, `6`, . . . , `n/2` is a divisor. If a divisor is found, `n` is not prime. A more efficient approach is to check whether any of the prime numbers less than or equal to $\sqrt{n}$ can divide `n` evenly. If not, `n` is prime. Rewrite Listing 5.15 to display the first 50 prime numbers using this approach. You need to use an array to store the prime numbers, and later use them to check whether they are possible divisors for `n`.

***7.7** (*Count single digits*) Write a program that generates 100 random integers between 0 and 9 and displays the count for each number. (*Hint*: Use an array of 10 integers, say `counts`, to store the counts for the number of 0s, 1s, . . . , 9s.)

Sections 7.6–7.8

7.8 (*Average an array*) Write two overloaded methods that return the average of an array with the following headers:

```
public static double average(int[] array)
public static double average(double[] array)
```

Write a test program that prompts the user to enter 10 integers, invokes the first method, then displays the average value; prompts the user to enter 10 double values, invokes the second method, then displays the average value.

7.9 (*Find the smallest element*) Write a method that finds the smallest element in an array of double values using the following header:

```
public static double min(double[] array)
```

Write a test program that prompts the user to enter 10 numbers, invokes this method to return the minimum value, and displays the minimum value. Here is a sample run of the program:

```
Enter 10 numbers: 1.9 2.5 3.7 2 1.5 6 3 4 5 2 ↵ Enter
The minimum number is 1.5
```

7.10 (*Find the index of the smallest element*) Write a method that returns the index of the smallest element in an array of integers. If the number of such elements is greater than 1, return the smallest index. Use the following header:

```
public static int indexOfSmallestElement(double[] array)
```

Write a test program that prompts the user to enter 10 numbers, invokes this method to return the index of the smallest element, and displays the index.

***7.11** (*Statistics: compute deviation*) Programming Exercise 5.45 computes the standard deviation of numbers. This exercise uses a different but equivalent formula to compute the standard deviation of `n` numbers.

$$\text{mean} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1 + x_2 + \cdots + x_n}{n} \qquad \text{deviation} = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \text{mean})^2}{n - 1}}$$

To compute the standard deviation with this formula, you have to store the individual numbers using an array, so they can be used after the mean is obtained.

Your program should contain the following methods:

```
/** Compute the deviation of double values */
public static double deviation(double[] x)

/** Compute the mean of an array of double values */
public static double mean(double[] x)
```

Write a test program that prompts the user to enter 10 numbers and displays the mean and standard deviation, as presented in the following sample run:

```
Enter 10 numbers: 1.9 2.5 3.7 2 1 6 3 4 5 2 ↵ Enter
The mean is 3.11
The standard deviation is 1.55738
```

***7.12** (*Reverse an array*) The `reverse` method in Section 7.7 reverses an array by copying it to a new array. Rewrite the method that reverses the array passed in the argument and returns this array. Write a test program that prompts the user to enter 10 numbers, invokes the method to reverse the numbers, and displays the numbers.

Section 7.9

***7.13** (Random number chooser) Write the following method that returns a random number between `start` and `end`, excluding the `numbers`.

```
public static int getRandom(int start, int end, int... numbers)
```

For example, invoking `getRandom(1,100,4,8,95,93)` returns a random number between `1` and `100` excluding `4,8,95,`and `93`. Write a test program that invokes `getRandom(1,100,4,8,95,93)` 45 times and displays the resulting numbers 15 per line using the format `%4d`.

7.14 (*Compute gcd*) Write a method that returns the gcd of an unspecified number of integers. The method header is specified as follows:

```
public static int gcd(int... numbers)
```

Write a test program that prompts the user to enter five numbers, invokes the method to find the gcd of these numbers, and displays the gcd.

Sections 7.10–7.12

7.15 (*Eliminate duplicates*) Write a method that returns a new array by eliminating the duplicate values in the array using the following method header:

```
public static int[] eliminateDuplicates(int[] list)
```

Write a test program that reads in 10 integers, invokes the method, and displays the distinct numbers separated by exactly one space. Here is a sample run of the program:

```
Enter 10 numbers: 1 2 3 2 1 6 3 4 5 2 ↵Enter
The distinct numbers are: 1 2 3 6 4 5
```

7.16 (*Execution time*) Write a program that randomly generates an array of 100,000 integers and a key. Estimate the execution time of invoking the `linearSearch` method in Listing 7.6. Sort the array and estimate the execution time of invoking the `binarySearch` method in Listing 7.7. You can use the following code template to obtain the execution time:

```
long startTime = System.nanoTime();
perform the task;
long endTime = System.nanoTime();
long executionTime = endTime - startTime;
```

****7.17** (*Sort students*) Write a program that prompts the user to enter the number of students, the students' names, and their scores and prints student names in decreasing order of their scores. Assume the name is a string without spaces, use the `Scanner`'s `next()` method to read a name.

****7.18** (*Bubble sort*) Write a sort method that uses the bubble-sort algorithm. The bubble-sort algorithm makes several passes through the array. On each pass, successive neighboring pairs are compared. If a pair is not in order, its values are swapped; otherwise, the values remain unchanged. The technique is called a *bubble sort* or *sinking sort* because the smaller values gradually "bubble" their way to the top, and the larger values "sink" to the bottom. Write a test program that reads in 10 double numbers, invokes the method, and displays the sorted numbers.

****7.19** (*Sorted?*) Write the following method that returns true if the list is already sorted in nondecreasing order:

```
public static boolean isSorted(int[] list)
```

Write a test program that prompts the user to enter a list and displays whether the list is sorted or not. Here is a sample run. Note that the program first prompts the user to enter the size of the list.

```
Enter the size of the list: 8 ↵Enter
Enter the contents of the list: 10 1 5 16 61 9 11 1 ↵Enter
The list has 8 integers 10 1 5 16 61 9 11 1
The list is not sorted
```

```
Enter the size of the list: 10 ↵Enter
Enter the contents of the list: 1 1 3 4 4 5 7 9 11 21 ↵Enter
The list has 10 integers 1 1 3 4 4 5 7 9 11 21
The list is already sorted
```

***7.20** (*Revise selection sort*) In Listing 7.8, you used selection sort to sort an array. The selection-sort method repeatedly finds the smallest number in the current array and swaps it with the first. Rewrite this program by finding the largest number and swapping it with the last. Write a test program that reads in 10 double numbers, invokes the method, and displays the sorted numbers.

Section 7.13

***7.21** (*Sum integers*) Write a program that passes an unspecified number of integers from command line and displays their total.

***7.22** (*Find the number of uppercase letters in a string*) Write a program that passes a string to the command line and displays the number of uppercase letters in the string.

Comprehensive

****7.23** (*Game: locker puzzle*) A school has 100 lockers and 100 students. All lockers are closed on the first day of school. As the students enter, the first student, denoted as S1, opens every locker. Then the second student, S2, begins with the second locker, denoted as L2, and closes every other locker. Student S3 begins with the third locker and changes every third locker (closes it if it was open and opens it if it was closed). Student S4 begins with locker L4 and changes every fourth locker. Student S5 starts with L5 and changes every fifth locker, and so on, until student S100 changes L100.

After all the students have passed through the building and changed the lockers, which lockers are open? Write a program to find your answer and display all open locker numbers separated by exactly one space.

(*Hint*: Use an array of 100 Boolean elements, each of which indicates whether a locker is open (`true`) or closed (`false`). Initially, all lockers are closed.)

VideoNote

Coupon collector's problem

****7.24** (*Simulation: coupon collector's problem*) Coupon collector is a classic statistics problem with many practical applications. The problem is to repeatedly pick objects from a set of objects and find out how many picks are needed for all the objects to be picked at least once. A variation of the problem is to pick cards from a shuffled deck of 52 cards repeatedly, and find out how many picks are needed before you see one of each suit. Assume a picked card is placed back in the deck before picking another. Write a program to simulate the number of picks needed to get four cards from each suit and display the four cards picked (it is possible a card may be picked twice). Here is a sample run of the program:

```
Queen of Spades
5 of Clubs
Queen of Hearts
4 of Diamonds
Number of picks: 12
```

7.25 (*Algebra: solve quadratic equations*) Write a method for solving a quadratic equation using the following header:

```
public static int solveQuadratic(double[] eqn, double[] roots)
```

The coefficients of a quadratic equation $ax^2 + bx + c = 0$ are passed to the array `eqn` and the real roots are stored in roots. The method returns the number of real roots. See Programming Exercise 3.1 on how to solve a quadratic equation.

Write a program that prompts the user to enter values for a, b, and c and displays the number of real roots and all real roots.

7.26 (*Strictly identical arrays*) The arrays `list1` and `list2` are *strictly identical* if their corresponding elements are equal. Write a method that returns `true` if `list1` and `list2` are strictly identical, using the following header:

```
public static boolean equals(int[] list1, int[] list2)
```

Write a test program that prompts the user to enter two lists of integers and displays whether the two are strictly identical. Here are the sample runs. Note the first number in the input for each list indicates the number of the elements in the list. This number is not part of the list.

```
Enter list1 size and contents: 5 2 5 6 1 6 ↵Enter
Enter list2 size and contents: 5 2 5 6 1 6 ↵Enter
Two lists are strictly identical
```

```
Enter list1 size and contents: 5 2 5 6 6 1 ↵Enter
Enter list2 size and contents: 5 2 5 6 1 6 ↵Enter
Two lists are not strictly identical
```

7.27 (*Identical arrays*) The arrays `list1` and `list2` are *identical* if they have the same contents. Write a method that returns `true` if `list1` and `list2` are identical, using the following header:

```
public static boolean equals(int[] list1, int[] list2)
```

Write a test program that prompts the user to enter two lists of integers and displays whether the two are identical. Here are the sample runs. Note the first number in the input for each list indicates the number of the elements in the list. This number is not part of the list.

```
Enter list1 size and contents: 5 2 5 6 6 1 ↵Enter
Enter list2 size and contents: 5 5 2 6 1 6 ↵Enter
Two lists are identical
```

```
Enter list1: 5 5 5 6 6 1 ↵Enter
Enter list2: 5 2 5 6 1 6 ↵Enter
Two lists are not identical
```

****7.28** (*Math: combinations*) Write a program that prompts the user to enter 10 integers and displays all combinations of picking two numbers from the 10 numbers.

****7.29** (*Game: pick four cards*) Write a program that picks four cards from a deck of 52 cards and computes their sum. An Ace, King, Queen, and Jack represent 1, 13, 12, and 11, respectively. Your program should display the number of picks that yields the sum of 24.

VideoNote

Consecutive four

***7.30** (*Pattern recognition: consecutive four equal numbers*) Write the following method that tests whether the array has four consecutive numbers with the same value:

```
public static boolean isConsecutiveFour(int[] values)
```

Write a test program that prompts the user to enter a series of integers and displays it if the series contains four consecutive numbers with the same value. Your program should first prompt the user to enter the input size—i.e., the number of values in the series. Here are sample runs:

```
Enter the number of values: 8 ↵Enter
Enter the values: 3 4 5 5 5 5 4 5 ↵Enter
The list has consecutive fours
```

```
Enter the number of values: 9 ↵Enter
Enter the values: 3 4 5 5 6 5 5 4 5 ↵Enter
The list has no consecutive fours
```

****7.31** (*Merge two sorted lists*) Write the following method that merges two sorted lists into a new sorted list:

```
public static int[] merge(int[] list1, int[] list2)
```

Implement the method in a way that takes at most `list1.length` + `list2.length` comparisons. See liveexample.pearsoncmg.com/dsanimation/MergeSortNeweBook.html for an animation of the implementation. Write a test program that prompts the user to enter two sorted lists and displays the merged list. Here is a sample run. Note the first number in the input indicates the number of the elements in the list. This number is not part of the list.

```
Enter list1 size and contents: 5 1 5 16 61 111 ↵Enter
Enter list2 size and contents: 4 2 4 5 6 ↵Enter
list1 is 1 5 16 61 111
list2 is 2 4 5 6
The merged list is 1 2 4 5 5 6 16 61 111
```

****7.32** (*Partition of a list*) Write the following method that partitions the list using the first element, called a *pivot*:

```
public static int partition(int[] list)
```

After the partition, the elements in the list are rearranged so all the elements before the pivot are less than or equal to the pivot, and the elements after the pivot are greater than the pivot. The method returns the index where the pivot is located in the new list. For example, suppose the list is {5, 2, 9, 3, 6, 8}. After the partition, the list becomes {3, 2, 5, 9, 6, 8}. Implement the method in a way that takes at most `list.length` comparisons. See liveexample.pearsoncmg.com/dsanimation/QuickSortNeweBook.html for an animation of the implementation. Write a test program that prompts the user to enter the size of the list and the contents of the list and displays the list after the partition. Here is a sample run.

```
Enter list size: 8 ↵Enter
Enter list content: 10 1 5 16 61 9 11 1 ↵Enter
After the partition, the list is 9 1 5 1 10 61 11 16
```

***7.33** (*Culture: Chinese Zodiac*) Simplify Listing 3.9 using an array of strings to store the animal names.

****7.34** (*Sort characters in a string*) Write a method that returns a sorted string using the following header:

```
public static String sort(String s)
```

For example, `sort("acb")` returns `abc`.

Write a test program that prompts the user to enter a string and displays the sorted string.

*****7.35** (*Game: hangman*) Write a hangman game that randomly generates a word and prompts the user to guess one letter at a time, as presented in the sample run. Each letter in the word is displayed as an asterisk. When the user makes a correct guess, the actual letter is then displayed. When the user finishes a word, display the number of misses and ask the user whether to continue to play with another word. Declare an array to store words, as follows:

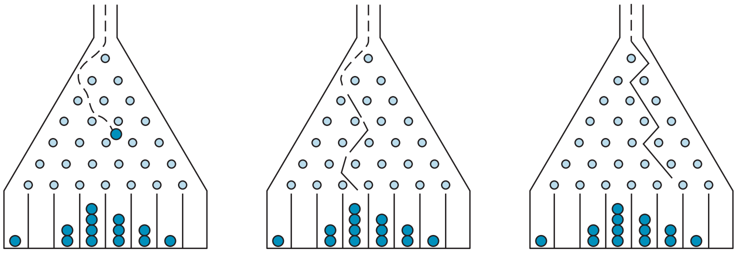
```
// Add any words you wish in this array
String[] words = {"write", "that",...};
```

```
(Guess) Enter a letter in word ******* > p ↵Enter
(Guess) Enter a letter in word p****** > r ↵Enter
(Guess) Enter a letter in word pr**r** > p ↵Enter
     p is already in the word
(Guess) Enter a letter in word pr**r** > o ↵Enter
(Guess) Enter a letter in word pro*r** > g ↵Enter
(Guess) Enter a letter in word progr** > n ↵Enter
     n is not in the word
(Guess) Enter a letter in word progr** > m ↵Enter
(Guess) Enter a letter in word progr*m > a ↵Enter
The word is program. You missed 1 time
Do you want to guess another word? Enter y or n>
```

*****7.36** (*Game: Eight Queens*) The classic Eight Queens puzzle is to place eight queens on a chessboard such that no two queens can attack each other (i.e., no two queens are on the same row, same column, or same diagonal). There are many possible solutions. Write a program that displays one such solution. A sample output is shown below:

```
|Q| | | | | | | |
| | | | |Q| | | |
| | | | | | | |Q|
| | | | | |Q| | |
| | |Q| | | | | |
| | | | | | |Q| |
| |Q| | | | | | |
| | | |Q| | | | |
```

*****7.37** (*Game: bean machine*) The bean machine, also known as a quincunx or the Galton box, is a device for statistics experiments named after English scientist Sir Francis Galton. It consists of an upright board with evenly spaced nails (or pegs) in a triangular form, as shown in Figure 7.13.



(a) (b) (c)

Balls are dropped from the opening of the board. Every time a ball hits a nail, it has a 50% chance of falling to the left or to the right. The piles of balls are accumulated in the slots at the bottom of the board.

Write a program that simulates the bean machine. Your program should prompt the user to enter the number of the balls and the number of the slots in the machine. Simulate the falling of each ball by printing its path. For example, the path for the ball in Figure 7.13b is LLRRLLR and the path for the ball in Figure 7.13c is RLRRLRR. Display the final buildup of the balls in the slots in a histogram. Here is a sample run of the program:

(*Hint*: Create an array named `slots`. Each element in `slots` stores the number of balls in a slot. Each ball falls into a slot via a path. The number of Rs in a path is the position of the slot where the ball falls. For example, for the path LRLRLRR, the ball falls into `slots[4]`, and for the path RRLLLLL, the ball falls into `slots[2]`.)

```
Enter the number of balls to drop: 5 ↵Enter
Enter the number of slots in the bean machine: 8 ↵Enter

LRLRLRR
RRLLLRR
LLRLLRR
RRLLLLL
LRLRRLR

    0
    0
  000
```

CHAPTER

8

MULTIDIMENSIONAL ARRAYS

Objectives

- To give examples of representing data using two-dimensional arrays (§8.1).
- To declare variables for two-dimensional arrays, create arrays, and access array elements in a two-dimensional array using row and column indices (§8.2).
- To program common operations for two-dimensional arrays (displaying arrays, summing all elements, finding the minimum and maximum elements, and random shuffling) (§8.3).
- To pass two-dimensional arrays to methods (§8.4).
- To write a program for grading multiple-choice questions using two-dimensional arrays (§8.5).
- To solve the closest pair problem using two-dimensional arrays (§8.6).
- To check a Sudoku solution using two-dimensional arrays (§8.7).
- To use multidimensional arrays (§8.8).

8.1 Introduction

Data in a table or a matrix can be represented using a two-dimensional array.

Key Point

problem

A *two-dimensional array* is an array that contains other arrays as its elements. The preceding chapter introduced how to use one-dimensional arrays to store linear collections of elements. You can use a two-dimensional array to store a matrix or a table. For example, the following table that lists the distances between cities can be stored using a two-dimensional array named **`distances`**.

Distance Table (in miles)

	Chicago	**Boston**	**New York**	**Atlanta**	**Miami**	**Dallas**	**Houston**
Chicago	0	983	787	714	1375	967	1087
Boston	983	0	214	1102	1763	1723	1842
New York	787	214	0	888	1549	1548	1627
Atlanta	714	1102	888	0	661	781	810
Miami	1375	1763	1549	661	0	1426	1187
Dallas	967	1723	1548	781	1426	0	239
Houston	1087	1842	1627	810	1187	239	0

```
double[][] distances = {
  {0, 983, 787, 714, 1375, 967, 1087},
  {983, 0, 214, 1102, 1763, 1723, 1842},
  {787, 214, 0, 888, 1549, 1548, 1627},
  {714, 1102, 888, 0, 661, 781, 810},
  {1375, 1763, 1549, 661, 0, 1426, 1187},
  {967, 1723, 1548, 781, 1426, 0, 239},
  {1087, 1842, 1627, 810, 1187, 239, 0},
};
```

Each element in the **`distances`** array is another array, so **`distances`** is considered a *nested array*. In this example, a two-dimensional array is used to store two-dimensional data.

8.2 Two-Dimensional Array Basics

An element in a two-dimensional array is accessed through a row and a column index.

Key Point

How do you declare a variable for two-dimensional arrays? How do you create a two-dimensional array? How do you access elements in a two-dimensional array? This section will address these issues.

8.2.1 Declaring Variables of Two-Dimensional Arrays and Creating Two-Dimensional Arrays

The syntax for declaring a two-dimensional array is as follows:

```
elementType[][] arrayRefVar;
```

or

```
elementType arrayRefVar[][]; // Allowed, but not preferred
```

As an example, here is how you would declare a two-dimensional array variable **`matrix`** of **`int`** values:

```
int[][] matrix;
```

or

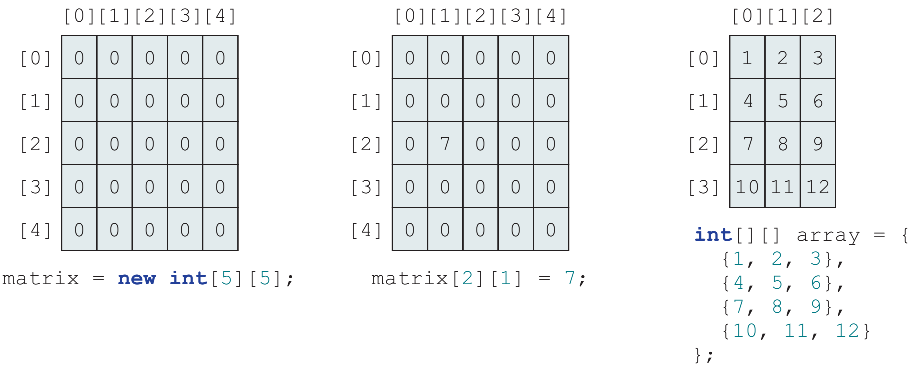
```
int matrix[][]; // This style is allowed, but not preferred
```

You can create a two-dimensional array of 5-by-5 `int` values and assign it to `matrix` using this syntax:

```
matrix = new int[5][5];
```

Two subscripts are used in a two-dimensional array: one for the row, and the other for the column. The two subscripts are conveniently called *row index* and *column index*. As in a one-dimensional array, the index for each subscript is of the `int` type and starts from `0`, as shown in Figure 8.1a.

row index
column index



	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	0	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

```
matrix = new int[5][5];
```

(a)

	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	7	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

```
matrix[2][1] = 7;
```

(b)

	[0]	[1]	[2]
[0]	1	2	3
[1]	4	5	6
[2]	7	8	9
[3]	10	11	12

```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

(c)

FIGURE 8.1 The index of each subscript of a two-dimensional array is an `int` value, starting from `0`.

To assign the value `7` to a specific element at row index `2` and column index `1`, as shown in Figure 8.1b, you can use the following syntax:

```
matrix[2][1] = 7;
```

Caution

It is a common mistake to use `matrix[2, 1]` to access the element at row `2` and column `1`. In Java, each subscript must be enclosed in a pair of square brackets.

You can also use an array initializer to declare, create, and initialize a two-dimensional array. For example, the following code in (a) creates an array with the specified initial values, as shown in Figure 8.1c. This is equivalent to the code in (b).

```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

(a)

Equivalent

```
int[][] array = new int[4][3];
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;
array[2][0] = 7; array[2][1] = 8; array[2][2] = 9;
array[3][0] = 10; array[3][1] = 11; array[3][2] = 12;
```

(b)

8.2.2 Obtaining the Lengths of Two-Dimensional Arrays

A two-dimensional array is actually an array in which each element is a one-dimensional array. The length of an array `x` is the number of elements in the array, which can be obtained using `x.length`. `x[0]`, `x[1]`, . . . , and `x[x.length - 1]` are arrays. Their lengths can be obtained using `x[0].length`, `x[1].length`, . . . , and `x[x.length - 1].length`.

For example, suppose that `x = new int[3][4]`, `x[0]`, `x[1]`, and `x[2]` are one-dimensional arrays and each contains four elements, as shown in Figure 8.2. `x.length` is `3`, and `x[0].length`, `x[1].length`, and `x[2].length` are `4`.

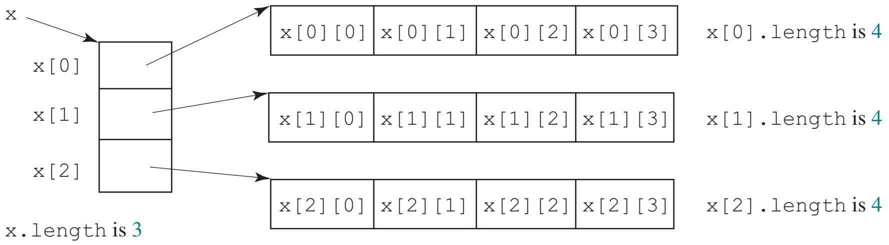
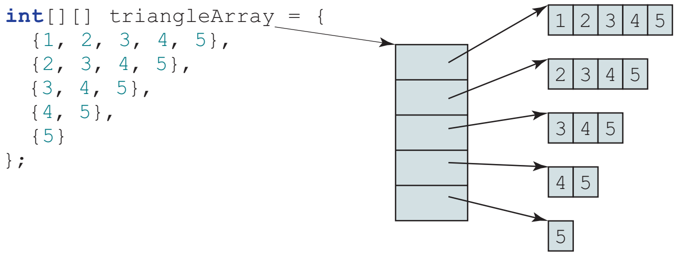

x
x[0]
x[1]
x[2]
x.length is 3
x[0][0] x[0][1] x[0][2] x[0][3]
x[1][0] x[1][1] x[1][2] x[1][3]
x[2][0] x[2][1] x[2][2] x[2][3]
x[0].length is 4
x[1].length is 4
x[2].length is 4

FIGURE 8.2 A two-dimensional array is a one-dimensional array in which each element is another one-dimensional array.

8.2.3 Ragged Arrays

ragged array

Each row in a two-dimensional array is itself an array. Thus, the rows can have different lengths. An array of this kind is known as a *ragged array*. Here is an example of creating a ragged array:



```
int[][] triangleArray = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```

As you can see, `triangleArray[0].length` is 5, `triangleArray[1].length` is 4, `triangleArray[2].length` is 3, `triangleArray[3].length` is 2, and `triangleArray[4].length` is 1.

If you don't know the values in a ragged array in advance, but do know the sizes—say, the same as in the preceding figure—you can create a ragged array using the following syntax:

```
int[][] triangleArray = new int[5][];
triangleArray[0] = new int[5];
triangleArray[1] = new int[4];
triangleArray[2] = new int[3];
triangleArray[3] = new int[2];
triangleArray[4] = new int[1];
```

You can now assign values to the array. For example,

```
triangleArray[0][3] = 4;
triangleArray[4][0] = 5;
```

Note

The syntax `new int[5][]` for creating an array requires the first index to be specified. The syntax `new int[][]` would be wrong.

Check Point

8.2.1 Declare an array reference variable for a two-dimensional array of `int` values, create a 4-by-5 `int` matrix, and assign it to the variable.

8.2.2 Which of the following statements are valid?

```
int[][] r = new int[2];
int[] x = new int[];
int[][] y = new int[3][];
int[][] z = {{1, 2}};
int[][] m = {{1, 2}, {2, 3}};
int[][] n = {{1, 2}, {2, 3}, };
```

8.2.3 Write an expression to obtain the row size of a two-dimensional array `x` and an expression to obtain the size of the first row.

8.2.4 Can the rows in a two-dimensional array have different lengths?

8.2.5 What is the output of the following code?

```
int[][] array = new int[5][6];
int[] x = {1, 2};
array[0] = x;
System.out.println("array[0][1] is " + array[0][1]);
```

8.3 Processing Two-Dimensional Arrays

Key Point

Nested `for` *loops are often used to process a two-dimensional array.*

Suppose an array `matrix` is created as follows:

```
int[][] matrix = new int[10][10];
```

The following are some examples of processing two-dimensional arrays.

1. *Initializing arrays with input values.* The following loop initializes the array with user input values:

```
java.util.Scanner input = new java.util.Scanner(System.in);
System.out.println("Enter " + matrix.length + " rows and " +
  matrix[0].length + " columns: ");
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    matrix[row][column] = input.nextInt();
  }
}
```

2. *Initializing arrays with random values.* The following loop initializes the array with random values between `0` and `99`:

```
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    matrix[row][column] = (int)(Math.random() *  100);
  }
}
```

3. *Printing arrays.* To print a two-dimensional array, you have to print each element in the array using a loop like the following loop:

```
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    System.out.print(matrix[row][column] + " ");
  }

  System.out.println();
}
```

4. *Summing all elements*. Use a variable named `total` to store the sum. Initially `total` is `0`. Add each element in the array to `total` using a loop like this:

```
int total = 0;
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    total += matrix[row][column];
  }
}
```

5. *Summing elements by column*. For each column, use a variable named `total` to store its sum. Add each element in the column to `total` using a loop like this:

```
for (int column = 0; column < matrix[0].length; column++) {
  int total = 0;
  for (int row = 0; row < matrix.length; row++)
    total += matrix[row][column];
  System.out.println("Sum for column " + column + " is "
    + total);
}
```

VideoNote

Find the row with the largest sum

6. *Which row has the largest sum?* Use variables `maxRow` and `indexOfMaxRow` to track the largest sum and index of the row. For each row, compute its sum and update `maxRow` and `indexOfMaxRow` if the new sum is greater.

```
int maxRow = 0;
int indexOfMaxRow = 0;

// Get sum of the first row in maxRow
for (int column = 0; column < matrix[0].length; column++) {
  maxRow += matrix[0][column];
}

for (int row = 1; row < matrix.length; row++) {
  int totalOfThisRow = 0;
  for (int column = 0; column < matrix[row].length; column++)
    totalOfThisRow += matrix[row][column];

  if (totalOfThisRow > maxRow) {
    maxRow = totalOfThisRow;
    indexOfMaxRow = row;
  }
}

System.out.println("Row " + indexOfMaxRow
  + " has the maximum sum of " + maxRow);
```

7. *Random shuffling*. Shuffling the elements in a one-dimensional array was introduced in Section 7.2.6. How do you shuffle all the elements in a two-dimensional array? To accomplish this, for each element `matrix[i][j]`, randomly generate indices `i1` and `j1` and swap `matrix[i][j]` with `matrix[i1][j1]`, as follows:

```
for (int i = 0; i < matrix.length; i++) {
  for (int j = 0; j < matrix[i].length; j++) {
    int i1 = (int)(Math.random() * matrix.length);
    int j1 = (int)(Math.random() * matrix[i].length);

    // Swap matrix[i][j] with matrix[i1][j1]
    int temp = matrix[i][j];
    matrix[i][j] = matrix[i1][j1];
    matrix[i1][j1] = temp;
  }
}
```

Check Point

8.3.1 Show the output of the following code:

```
int[][] array = {{1, 2}, {3, 4}, {5, 6}};
for (int i = array.length – 1; i >= 0; i––) {
  for (int j = array[i].length – 1; j >= 0; j––)
    System.out.print(array[i][j] + " ");
  System.out.println();
}
```

8.3.2 Show the output of the following code:

```
int[][] array = {{1, 2}, {3, 4}, {5, 6}};
int sum = 0;
for (int i = 0; i < array.length; i++)
  sum += array[i][0];
System.out.println(sum);
```

8.4 Passing Two-Dimensional Arrays to Methods

Key Point

When passing a two-dimensional array to a method, the reference of the array is passed to the method.

You can pass a two-dimensional array to a method just as you pass a one-dimensional array. You can also return an array from a method. Listing 8.1 gives an example with two methods. The first method, `getArray()`, returns a two-dimensional array and the second method, `sum(int[][] m)`, returns the sum of all the elements in a matrix.

Listing 8.1 PassTwoDimensionalArray.java

```
import java.util.Scanner;

public class PassTwoDimensionalArray {
  public static void main(String[] args) {
    int[][] m = getArray(); // Get an array

    // Display sum of elements
    System.out.println("\nSum of all elements is " + sum(m));
  }

  public static int[][] getArray() {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Enter array values
    int[][] m = new int[3][4];
    System.out.println("Enter " + m.length + " rows and "
      + m[0].length + " columns: ");
    for (int i = 0; i < m.length; i++)
      for (int j = 0; j < m[i].length; j++)
        m[i][j] = input.nextInt();

```

get array

pass array

`getArray` method

return array

sum method

```
23      return m;
24    }
25
26    public static int sum(int[][] m) {
27      int total = 0;
28      for (int row = 0; row < m.length; row++) {
29        for (int column = 0; column < m[row].length; column++) {
30          total += m[row][column];
31        }
32      }
33
34      return total;
35    }
36  }
```

```
Enter 3 rows and 4 columns:
1 2 3 4 ↵Enter
5 6 7 8 ↵Enter
9 10 11 12 ↵Enter

Sum of all elements is 78
```

The method **getArray** prompts the user to enter values for the array (lines 11–24) and returns the array (line 23).

The method **sum** (lines 26–35) has a two-dimensional array argument. You can obtain the number of rows using `m.length` (line 28), and the number of columns in a specified row using `m[row].length` (line 29).

Check Point

8.4.1 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    int[][] array = {{1, 2, 3, 4}, {5, 6, 7, 8}};
    System.out.println(m1(array)[0]);
    System.out.println(m1(array)[1]);
  }

  public static int[] m1(int[][] m) {
    int[] result = new int[2];
    result[0] = m.length;
    result[1] = m[0].length;
    return result;
  }
}
```

8.5 Case Study: Grading a Multiple-Choice Test

Key Point

The problem is to write a program that will grade multiple-choice tests.

VideoNote
Grade multiple-choice test

Suppose you need to write a program that grades multiple-choice tests. Assume there are eight students and ten questions, and the answers are stored in a two-dimensional array. Each row records a student's answers to the questions, as shown in the following array:

Students' Answers to the Questions:

	0	1	2	3	4	5	6	7	8	9
Student 0	A	B	A	C	C	D	E	E	A	D
Student 1	D	B	A	B	C	A	E	E	A	D
Student 2	E	D	D	A	C	B	E	E	A	D
Student 3	C	B	A	E	D	C	E	E	A	D
Student 4	A	B	D	C	C	D	E	E	A	D
Student 5	B	B	E	C	C	D	E	E	A	D
Student 6	B	B	A	C	C	D	E	E	A	D
Student 7	E	B	E	C	C	D	E	E	A	D

The key is stored in a one-dimensional array:

Key to the Questions:

	0	1	2	3	4	5	6	7	8	9
Key	D	B	D	C	C	D	A	E	A	D

Your program grades the test and displays the result. It compares each student's answers with the key, counts the number of correct answers, and displays it. Listing 8.2 gives the program.

LISTING 8.2 GradeExam.java

```java
public class GradeExam {
  /** Main method */
  public static void main(String[] args) {
    // Students' answers to the questions
    char[][] answers = {
      {'A', 'B', 'A', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
      {'D', 'B', 'A', 'B', 'C', 'A', 'E', 'E', 'A', 'D'},
      {'E', 'D', 'D', 'A', 'C', 'B', 'E', 'E', 'A', 'D'},
      {'C', 'B', 'A', 'E', 'D', 'C', 'E', 'E', 'A', 'D'},
      {'A', 'B', 'D', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
      {'B', 'B', 'E', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
      {'B', 'B', 'A', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
      {'E', 'B', 'E', 'C', 'C', 'D', 'E', 'E', 'A', 'D'}};

    // Key to the questions
    char[] keys = {'D', 'B', 'D', 'C', 'C', 'D', 'A', 'E', 'A', 'D'};

    // Grade all answers
    for (int i = 0; i < answers.length; i++) {
      // Grade one student
      int correctCount = 0;
      for (int j = 0; j < answers[i].length; j++) {
        if (answers[i][j] == keys[j])
          correctCount++;
      }

      System.out.println("Student " + i + "'s correct count is " +
        correctCount);
    }
  }
}
```

2-D array

1-D array

compare with key

```
Student 0's correct count is 7
Student 1's correct count is 6
Student 2's correct count is 5
Student 3's correct count is 4
Student 4's correct count is 8
Student 5's correct count is 7
Student 6's correct count is 7
Student 7's correct count is 7
```

The statement in lines 5–13 declares, creates, and initializes a two-dimensional array of characters and assigns the reference to `answers` of the `char[][]` type.

The statement in line 16 declares, creates, and initializes an array of `char` values and assigns the reference to `keys` of the `char[]` type.

Each row in the array `answers` stores a student's answer, which is graded by comparing it with the key in the array `keys`. The result is displayed immediately after a student's answer is graded.

Check Point

8.5.1 How do you modify the code so it also displays the highest count and the student with the highest count?

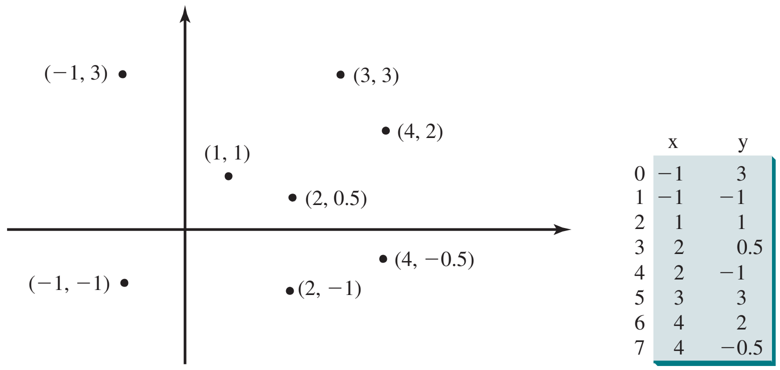
8.6 Case Study: Finding the Closest Pair

This section presents a geometric problem for finding the closest pair of points.

Key Point

closest-pair animation on the Companion Website

Given a set of points, the closest-pair problem is to find the two points that are nearest to each other. In Figure 8.3, for example, points `(1, 1)` and `(2, 0.5)` are closest to each other. There are several ways to solve this problem. An intuitive approach is to compute the distances between all pairs of points and find the one with the minimum distance, as implemented in Listing 8.3.



	x	y
0	−1	3
1	−1	−1
2	1	1
3	2	0.5
4	2	−1
5	3	3
6	4	2
7	4	−0.5

FIGURE 8.3 Points can be represented in a two-dimensional array.

LISTING 8.3 FindNearestPoints.java

number of points

```
import java.util.Scanner;

public class FindNearestPoints {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter the number of points: ");
    int numberOfPoints = input.nextInt();

    // Create an array to store points
10      double[][] points = new double[numberOfPoints][2];
11      System.out.print("Enter " + numberOfPoints + " points: ");
12      for (int i = 0; i < points.length; i++) {
13        points[i][0] = input.nextDouble();
14        points[i][1] = input.nextDouble();
15      }
16
17      // p1 and p2 are the indices in the points' array
18      int p1 = 0, p2 = 1; // Initial two points
19      double shortestDistance = distance(points[p1][0], points[p1][1],
20        points[p2][0], points[p2][1]); // Initialize shortestDistance
21
22      // Compute distance for every two points
23      for (int i = 0; i < points.length; i++) {
24        for (int j = i + 1; j < points.length; j++) {
25          double distance = distance(points[i][0], points[i][1],
26            points[j][0], points[j][1]); // Find distance
27
28          if (shortestDistance > distance) {
29            p1 = i; // Update p1
30            p2 = j; // Update p2
31            shortestDistance = distance; // Update shortestDistance
32          }
33        }
34      }
35
36      // Display result
37      System.out.println("The closest two points are " +
38        "(" + points[p1][0] + ", " + points[p1][1] + ") and (" +
39        points[p2][0] + ", " + points[p2][1] + ")");
40    }
41
42    /** Compute the distance between two points (x1, y1) and (x2, y2)*/
43    public static double distance(
44        double x1, double y1, double x2, double y2) {
45      return Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
46    }
47  }
```

2-D array

read points

track two points

track `shortestDistance`

for each point `i`

for each point `j`

distance between `i` and `j`

distance between two points

update `shortestDistance`

```
Enter the number of points: 8 ↵Enter
Enter 8 points: -1 3 -1 -1 1 1 2 0.5 2 -1 3 3 4 2 4 -0.5 ↵Enter
The closest two points are (1, 1) and (2, 0.5)
```

The program prompts the user to enter the number of points (lines 6 and 7). The points are read from the console and stored in a two-dimensional array named `points` (lines 12–15). The program uses the variable `shortestDistance` (line 19) to store the distance between the two nearest points, and the indices of these two points in the `points` array are stored in `p1` and `p2` (line 18).

For each point at index `i`, the program computes the distance between `points[i]` and `points[j]` for all `j > i` (lines 23–34). Whenever a shorter distance is found, the variable `shortestDistance` and `p1` and `p2` are updated (lines 28–32).

The distance between two points `(x1, y1)` and `(x2, y2)` can be computed using the formula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ (lines 43–46).

The program assumes the plane has at least two points. You can easily modify the program to handle the case if the plane has zero or one point.

multiple closest pairs

Note that there might be more than one closest pair of points with the same minimum distance. The program finds one such pair. You may modify the program to find all closest pairs in Programming Exercise 8.8.

input file

Tip

It is cumbersome to enter all points from the keyboard. You may store the input in a file, say **FindNearestPoints.txt**, and run the program using the following command:

```
java FindNearestPoints < FindNearestPoints.txt
```

Check Point

8.6.1 What happens if the input has only one point?

8.7 Case Study: Sudoku

Key Point

The problem is to check whether a given Sudoku solution is correct.

VideoNote
Sudoku

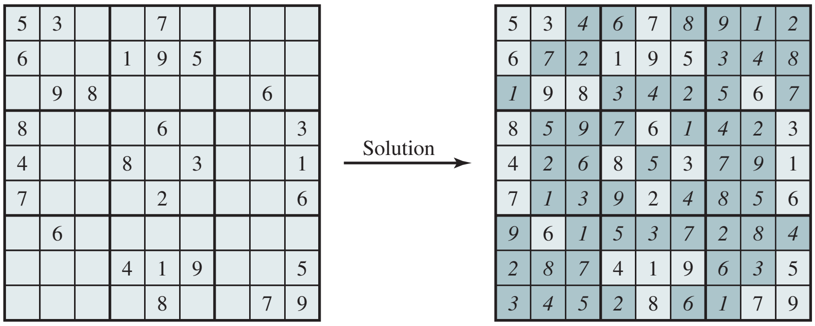
This section presents an interesting problem of a sort that appears in the newspaper every day. It is a number-placement puzzle, commonly known as *Sudoku*.

Writing a program to solve a Sudoku problem is very challenging. To make it accessible to the novice, this section presents a simplified version of the Sudoku problem, which is to verify whether a Sudoku solution is correct. The complete program for finding a Sudoku solution is presented in Supplement VI.C.

fixed cells

free cells

Sudoku is a 9 × 9 grid divided into smaller 3 × 3 boxes (also called *regions* or *blocks*), as shown in Figure 8.4a. Some cells, called *fixed cells*, are populated with numbers from **1** to **9**. The objective is to fill the empty cells, also called *free cells*, with the numbers **1** to **9** so every row, every column, and every 3 × 3 box contains the numbers **1** to **9**, as shown in Figure 8.4b.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6							
			4	1	9			5
				8			7	9

(a) Puzzle

5	3	*4*	*6*	7	*8*	*9*	*1*	*2*
6	*7*	*2*	1	9	5	*3*	*4*	*8*
1	9	8	*3*	*4*	*2*	*5*	6	*7*
8	*5*	*9*	*7*	6	*1*	*4*	*2*	3
4	*2*	*6*	8	*5*	3	*7*	*9*	1
7	*1*	*3*	*9*	2	*4*	*8*	*5*	6
9	6	*1*	*5*	*3*	*7*	*2*	*8*	*4*
2	*8*	*7*	4	1	9	*6*	*3*	5
3	*4*	*5*	*2*	8	*6*	*1*	7	9

(b) Solution

FIGURE 8.4 The Sudoku puzzle in (a) is solved in (b).

representing a grid

For convenience, we use value **0** to indicate a free cell, as shown in Figure 8.5a. The grid can be naturally represented using a two-dimensional array, as shown in Figure 8.5b.

5	3	0	0	7	0	0	0	0
6	0	0	1	9	5	0	0	0
0	9	8	0	0	0	0	6	0
8	0	0	0	6	0	0	0	3
4	0	0	8	0	3	0	0	1
7	0	0	0	2	0	0	0	6
0	6	0	0	0	0	0	0	0
0	0	0	4	1	9	0	0	5
0	0	0	0	8	0	0	7	9

(a)

```
int[][] grid =
  {{5, 3, 0, 0, 7, 0, 0, 0, 0},
   {6, 0, 0, 1, 9, 5, 0, 0, 0},
   {0, 9, 8, 0, 0, 0, 0, 6, 0},
   {8, 0, 0, 0, 6, 0, 0, 0, 3},
   {4, 0, 0, 8, 0, 3, 0, 0, 1},
   {7, 0, 0, 0, 2, 0, 0, 0, 6},
   {0, 6, 0, 0, 0, 0, 2, 8, 0},
   {0, 0, 0, 4, 1, 9, 0, 0, 5},
   {0, 0, 0, 0, 8, 0, 0, 7, 9}
  };
```

(b)

FIGURE 8.5 A grid can be represented using a two-dimensional array.

To find a solution for the puzzle, we must replace each **0** in the grid with an appropriate number from **1** to **9**. For the solution to the puzzle in Figure 8.5, the grid should be as shown in Figure 8.6.

Once a solution to a Sudoku puzzle is found, how do you verify that it is correct? Here are two approaches:

1. Check if every row has numbers from **1** to **9**, every column has numbers from **1** to **9**, and every small box has numbers from **1** to **9**.
2. Check each cell. Each cell must be a number from **1** to **9** and the cell must be unique on every row, every column, and every small box.

```
A solution grid is
   {{5, 3, 4, 6, 7, 8, 9, 1, 2},
    {6, 7, 2, 1, 9, 5, 3, 4, 8},
    {1, 9, 8, 3, 4, 2, 5, 6, 7},
    {8, 5, 9, 7, 6, 1, 4, 2, 3},
    {4, 2, 6, 8, 5, 3, 7, 9, 1},
    {7, 1, 3, 9, 2, 4, 8, 5, 6},
    {9, 6, 1, 5, 3, 7, 2, 8, 4},
    {2, 8, 7, 4, 1, 9, 6, 3, 5},
    {3, 4, 5, 2, 8, 6, 1, 7, 9}
   };
```

FIGURE 8.6 A solution is stored in `grid`.

The program in Listing 8.4 prompts the user to enter a solution and reports whether it is valid. We use the second approach in the program to check whether the solution is correct.

LISTING 8.4 CheckSudokuSolution.java

```java
import java.util.Scanner;

public class CheckSudokuSolution {
  public static void main(String[] args) {
    // Read a Sudoku solution
    int[][] grid = readASolution();

    System.out.println(isValid(grid) ? "Valid solution" :
      "Invalid solution");
  }

  /** Read a Sudoku solution from the console */
  public static int[][] readASolution() {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    System.out.println("Enter a Sudoku puzzle solution:");
    int[][] grid = new int[9][9];
    for (int i = 0; i < 9; i++)
      for (int j = 0; j < 9; j++)
        grid[i][j] = input.nextInt();

    return grid;
  }

  /** Check whether a solution is valid */
  public static boolean isValid(int[][] grid) {
```

read input

solution valid?

read solution

check solution

check rows

check columns

check small boxes

```
28      for (int i = 0; i < 9; i++)
29        for (int j = 0; j < 9; j++)
30          if (grid[i][j] < 1 || grid[i][j] > 9
31              || !isValid(i, j, grid))
32            return false;
33        return true; // The solution is valid
34    }
35
36    /** Check whether grid[i][j] is valid in the grid */
37    public static boolean isValid(int i, int j, int[][] grid) {
38      // Check whether grid[i][j] is unique in i's row
39      for (int column = 0; column < 9; column++)
40        if (column != j && grid[i][column] == grid[i][j])
41          return false;
42
43      // Check whether grid[i][j] is unique in j's column
44      for (int row = 0; row < 9; row++)
45        if (row != i && grid[row][j] == grid[i][j])
46          return false;
47
48      // Check whether grid[i][j] is unique in the 3-by-3 box
49      for (int row = (i / 3) * 3; row < (i / 3) * 3 + 3; row++)
50        for (int col = (j /  3) *  3; col < (j /  3) *  3 + 3; col++)
51          if (!(row == i && col == j) && grid[row][col] == grid[i][j])
52            return false;
53
54      return true; // The current value at grid[i][j] is valid
55    }
56  }
```

```
Enter a Sudoku puzzle solution:
9 6 3 1 7 4 2 5 8 ↵Enter
1 7 8 3 2 5 6 4 9 ↵Enter
2 5 4 6 8 9 7 3 1 ↵Enter
8 2 1 4 3 7 5 9 6 ↵Enter
4 9 6 8 5 2 3 1 7 ↵Enter
7 3 5 9 6 1 8 2 4 ↵Enter
5 8 9 7 1 3 4 6 2 ↵Enter
3 1 7 2 4 6 9 8 5 ↵Enter
6 4 2 5 9 8 1 7 3 ↵Enter
Valid solution
```

The program invokes the **`readASolution()`** method (line 6) to read a Sudoku solution and return a two-dimensional array representing a Sudoku grid.

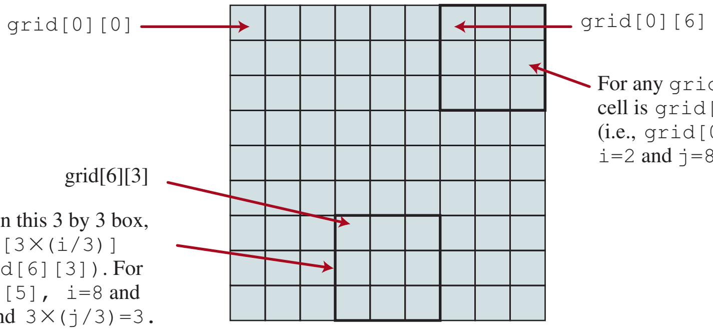
`isValid` method

The **`isValid(grid)`** method checks whether the values in the grid are valid by verifying that each value is between **`1`** and **`9`**, and that each value is valid in the grid (lines 27–34).

overloaded `isValid` method

The **`isValid(i, j, grid)`** method checks whether the value at **`grid[i][j]`** is valid. It checks whether **`grid[i][j]`** appears more than once in row **`i`** (lines 39–41), in column **`j`** (lines 44–46), and in the 3 × 3 box (lines 49–52).

How do you locate all the cells in the same box? For any **`grid[i][j]`**, the starting cell of the 3 × 3 box that contains it is **`grid[(i / 3) * 3][(j / 3) * 3]`**, as illustrated in Figure 8.7.



For any `grid[i][j]` in this 3 by 3 box, its starting cell is `grid[3×(i/3)][3×(j/3)]` (i.e., `grid[0][6]`). For example, for `grid[2][8]`, `i=2` and `j=8`, `3×(i/3)=0` and `3×(j/3)=6`.

For any `grid[i][j]` in this 3 by 3 box, its starting cell is `grid[3×(i/3)][3×(j/3)]` (i.e., `grid[6][3]`). For example, for `grid[8][5]`, `i=8` and `j=5`, `3×(i/3)=6` and `3×(j/3)=3`.

FIGURE 8.7 The location of the first cell in a 3 × 3 box determines the locations of other cells in the box.

With this observation, you can easily identify all the cells in the box. For instance, if **`grid[r][c]`** is the starting cell of a 3 × 3 box, the cells in the box can be traversed in a nested loop as follows:

```
// Get all cells in a 3-by-3 box starting at grid[r][c]
for (int row = r; row < r + 3; row++)
  for (int col = c; col < c + 3; col++)
    // grid[row][col] is in the box
```

It is cumbersome to enter 81 numbers from the console. When you test the program, you may store the input in a file, say **CheckSudokuSolution.txt** (see liveexample.pearsoncmg.com/data/CheckSudokuSolution.txt) and run the program using the following command:

input file

```
java CheckSudokuSolution < CheckSudokuSolution.txt
```

Check Point

8.7.1 What happens if the code in line 51 in Listing 8.4 is changed to

```
if (row != i && col != j && grid[row][col] == grid[i][j])
```

8.8 Multidimensional Arrays

A two-dimensional array is an array of one-dimensional arrays, and a three-dimensional array is an array of two-dimensional arrays.

Key Point

In the preceding section, you used a two-dimensional array to represent a matrix or a table. Occasionally, you will need to represent *n*-dimensional data structures. In Java, you can create *n*-dimensional arrays for any positive integer *n*.

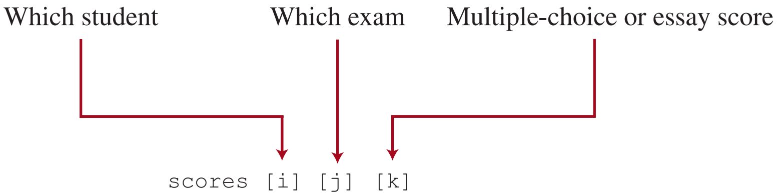
The way to declare two-dimensional array variables and create two-dimensional arrays can be generalized to declare *n*-dimensional array variables and create *n*-dimensional arrays for $n >= 3$. For example, you may use a three-dimensional array to store exam scores for a class of six students with five exams, and each exam has two parts (multiple-choice and essay type questions). The following syntax declares a three-dimensional array variable **`scores`**, creates an array, and assigns its reference to **`scores`**.

```
double[][][] scores = new double[6][5][2];
```

You can also use the array initializer to create and initialize the array as follows:

```
double[][][] scores = {
  {{7.5, 20.5}, {9.0, 22.5}, {15, 33.5}, {13, 21.5}, {15, 2.5}},
  {{4.5, 21.5}, {9.0, 22.5}, {15, 34.5}, {12, 20.5}, {14, 9.5}},
  {{6.5, 30.5}, {9.4, 10.5}, {11, 33.5}, {11, 23.5}, {10, 2.5}},
  {{6.5, 23.5}, {9.4, 32.5}, {13, 34.5}, {11, 20.5}, {16, 7.5}},
  {{8.5, 26.5}, {9.4, 52.5}, {13, 36.5}, {13, 24.5}, {16, 2.5}},
  {{9.5, 20.5}, {9.4, 42.5}, {13, 31.5}, {12, 20.5}, {16, 6.5}}};
```

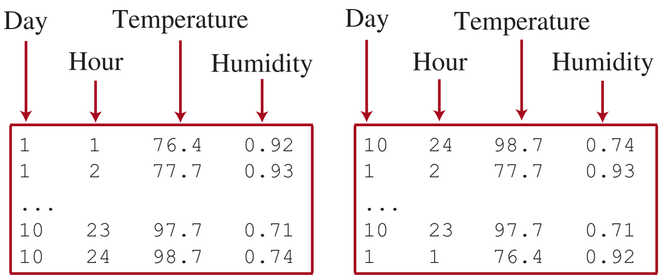
`scores[0][1][0]` refers to the multiple-choice score for the first student's second exam, which is `9.0`. `scores[0][1][1]` refers to the essay score for the first student's second exam, which is `22.5`. This is depicted in the following figure:



A multidimensional array is actually an array in which each element is another array. A three-dimensional array is an array of two-dimensional arrays. A two-dimensional array is an array of one-dimensional arrays. For example, suppose that `x = new int[2][2][5]` and `x[0]` and `x[1]` are two-dimensional arrays. `x[0][0]`, `x[0][1]`, `x[1][0]`, and `x[1][1]` are one-dimensional arrays and each contains five elements. `x.length` is `2`, `x[0].length` and `x[1].length` are `2`, and `x[0][0].length`, `x[0][1].length`, `x[1][0].length`, and `x[1][1].length` are `5`.

8.8.1 Case Study: Daily Temperature and Humidity

Suppose a meteorology station records the temperature and humidity every hour of every day, and stores the data for the past 10 days in a text file named **Weather.txt** (see liveexample.pearsoncmg.com/data/Weather.txt). Each line of the file consists of four numbers that indicate the day, hour, temperature, and humidity. The contents of the file may look like those in (a).



```
1     1     76.4    0.92
1     2     77.7    0.93
...
10    23    97.7    0.71
10    24    98.7    0.74
```

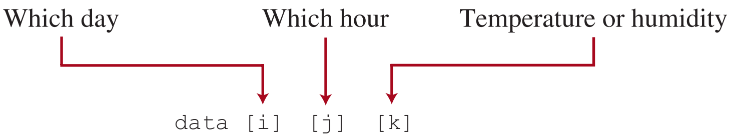
(a)

```
10    24    98.7    0.74
1     2     77.7    0.93
...
10    23    97.7    0.71
1     1     76.4    0.92
```

(b)

Note the lines in the file are not necessarily in increasing order of day and hour. For example, the file may appear as shown in (b).

Your task is to write a program that calculates the average daily temperature and humidity for the `10` days. You can use the input redirection to read the file and store the data in a three-dimensional array named `data`. The first index of `data` ranges from `0` to `9` and represents `10` days, the second index ranges from `0` to `23` and represents `24` hours, and the third index ranges from `0` to `1` and represents temperature and humidity, as depicted in the following figure:



Note the days are numbered from `1` to `10` and the hours from `1` to `24` in the file. Because the array index starts from `0`, `data[0][0][0]` stores the temperature in day `1` at hour `1` and `data[9][23][1]` stores the humidity in day `10` at hour `24`.

The program is given in Listing 8.5.

LISTING 8.5 Weather.java

```
 1  import java.util.Scanner;
 2
 3  public class Weather {
 4    public static void main(String[] args) {
 5      final int NUMBER_OF_DAYS = 10;
 6      final int NUMBER_OF_HOURS = 24;
 7      double[][][] data
 8        = new double[NUMBER_OF_DAYS][NUMBER_OF_HOURS][2];
 9
10      Scanner input = new Scanner(System.in);
11      // Read input using input redirection from a file
12      for (int k = 0; k < NUMBER_OF_DAYS * NUMBER_OF_HOURS; k++) {
13        int day = input.nextInt();
14        int hour = input.nextInt();
15        double temperature = input.nextDouble();
16        double humidity = input.nextDouble();
17        data[day - 1][hour - 1][0] = temperature;
18        data[day - 1][hour - 1][1] = humidity;
19      }
20
21      // Find the average daily temperature and humidity
22      for (int i = 0; i < NUMBER_OF_DAYS; i++) {
23        double dailyTemperatureTotal = 0, dailyHumidityTotal = 0;
24        for (int j = 0; j < NUMBER_OF_HOURS; j++) {
25          dailyTemperatureTotal += data[i][j][0];
26          dailyHumidityTotal += data[i][j][1];
27        }
28
29        // Display result
30        System.out.println("Day " + i + "'s average temperature is "
31          + dailyTemperatureTotal / NUMBER_OF_HOURS);
32        System.out.println("Day " + i + "'s average humidity is "
33          + dailyHumidityTotal / NUMBER_OF_HOURS);
34      }
35    }
36  }
```

three-dimensional array

```
Day 0's average temperature is 77.7708
Day 0's average humidity is 0.929583
Day 1's average temperature is 77.3125
Day 1's average humidity is 0.929583
...
Day 9's average temperature is 79.3542
Day 9's average humidity is 0.9125
```

You can use the following command to run the program:

```
java Weather < Weather.txt
```

A three-dimensional array for storing temperature and humidity is created in line 8. The loop in lines 12–19 reads the input to the array. You can enter the input from the keyboard, but doing so will be awkward. For convenience, we store the data in a file and use input redirection to read the data from the file. The loop in lines 24–27 adds all temperatures for each hour in a day to **`dailyTemperatureTotal`**, and all humidity for each hour to **`dailyHumidityTotal`**. The average daily temperature and humidity are displayed in lines 30–33.

8.8.2 Case Study: Guessing Birthdays

Listing 4.3, GuessBirthday.java, gives a program that guesses a birthday. The program can be simplified by storing the numbers in five sets in a three-dimensional array and it prompts the user for the answers using a loop, as given in Listing 8.6. The sample run of the program can be the same as given in Listing 4.3.

LISTING 8.6 `GuessBirthdayUsingArray.java`

```java
import java.util.Scanner;

public class GuessBirthdayUsingArray {
  public static void main(String[] args) {
    int day = 0; // Day to be determined
    int answer;

    int[][][] dates = {
      {{ 1,  3,  5,  7},
       { 9, 11, 13, 15},
       {17, 19, 21, 23},
       {25, 27, 29, 31}},
      {{ 2,  3,  6,  7},
       {10, 11, 14, 15},
       {18, 19, 22, 23},
       {26, 27, 30, 31}},
      {{ 4,  5,  6,  7},
       {12, 13, 14, 15}},
       {20, 21, 22, 23},
       {28, 29, 30, 31}},
      {{ 8,  9, 10, 11},
       {12, 13, 14, 15},
       {24, 25, 26, 27},
       {28, 29, 30, 31}},
      {{16, 17, 18, 19},
       {20, 21, 22, 23},
       {24, 25, 26, 27},
       {28, 29, 30, 31}}};

    // Create a Scanner
    Scanner input = new Scanner(System.in);

    for (int i = 0; i < 5; i++) {
      System.out.println("Is your birthday in Set" + (i + 1) + "?");
      for (int j = 0; j < 4; j++) {
        for (int k = 0; k < 4; k++)
          System.out.printf("%4d", dates[i][j][k]);
        System.out.println();
      }

      System.out.print("\nEnter 0 for No and 1 for Yes: ");
      answer = input.nextInt();

      if (answer == 1)
        day += dates[i][0][0];
```

three-dimensional array

Set `i`

add to `day`

```
46        }
47
48        System.out.println("Your birthday is " + day);
49      }
50    }
```

A three-dimensional array `dates` is created in lines 8–28. This array stores five sets of numbers. Each set is a 4-by-4 two-dimensional array.

The loop starting from line 33 displays the numbers in each set and prompts the user to answer whether the birthday is in the set (lines 41 and 42). If the day is in the set, the first number (`dates[i][0][0]`) in the set is added to variable `day` (line 45).

Check Point

8.8.1 Declare an array variable for a three-dimensional array, create a 4 × 6 × 5 `int` array, and assign its reference to the variable.

8.8.2 Assume `char[][][] x = new char[12][5][2]`, how many elements are in the array? What are `x.length`, `x[2].length`, and `x[0][0].length`?

8.8.3 Show the output of the following code:

```
int[][][] array = {{{1, 2}, {3, 4}}, {{5, 6},{7, 8}}};
System.out.println(array[0][0][0]);
System.out.println(array[1][1][1]);
```

KEY TERMS

column index 290
multidimensional array 303
nested array 290
row index 291
two-dimensional array 290

CHAPTER SUMMARY

1. A two-dimensional array can be used to store a table.
2. A variable for two-dimensional arrays can be declared using the syntax: `elementType[][] arrayVar`.
3. A two-dimensional array can be created using the syntax: `new elementType[ROW_SIZE][COLUMN_SIZE]`.
4. Each element in a two-dimensional array is represented using the syntax: `arrayVar[rowIndex][columnIndex]`.
5. You can create and initialize a two-dimensional array using an array initializer with the syntax: `elementType[][] arrayVar = {{row values}, . . ., {row values}}`.
6. You can use arrays of arrays to form multidimensional arrays. For example, a variable for three-dimensional arrays can be declared as `elementType[][][] arrayVar` and a three-dimensional array can be created using `new elementType[size1][size2][size3]`.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

***8.1** (*Sum elements column by column*) Write a method that returns the sum of all the elements in a specified column in a matrix using the following header:

```
public static double sumColumn(double[][] m, int columnIndex)
```

Write a test program that reads a 3-by-4 matrix and displays the sum of each column. Here is a sample run:

```
Enter a 3-by-4 matrix row by row:
1.5 2 3 4 ↵Enter
5.5 6 7 8 ↵Enter
9.5 1 3 1 ↵Enter
Sum of the elements at column 0 is 16.5
Sum of the elements at column 1 is 9.0
Sum of the elements at column 2 is 13.0
Sum of the elements at column 3 is 13.0
```

***8.2** (*Sum the major diagonal in a matrix*) Write a method that sums all the numbers in the major diagonal in an $n \times n$ matrix of `double` values using the following header:

```
public static double sumMajorDiagonal(double[][] m)
```

Write a test program that reads a 4-by-4 matrix and displays the sum of all its elements on the major diagonal. Here is a sample run:

```
Enter a 4-by-4 matrix row by row:
1 2 3 4.0 ↵Enter
5 6.5 7 8 ↵Enter
9 10 11 12 ↵Enter
13 14 15 16 ↵Enter
Sum of the elements in the major diagonal is 34.5
```

***8.3** (*Sort students on grades*) Rewrite Listing 8.2, GradeExam.java, to display the students in increasing order of the number of correct answers.

****8.4** (*Compute the weekly hours for each employee*) Suppose the weekly hours for all employees are stored in a two-dimensional array. Each row records an employee's seven-day work hours with seven columns. For example, the following array stores the work hours for eight employees. Write a program that displays employees and their total hours in decreasing order of the total hours.

	Su	M	T	W	Th	F	Sa
Employee 0	2	4	3	4	5	8	8
Employee 1	7	3	4	3	3	4	4
Employee 2	3	3	4	3	3	2	2
Employee 3	9	3	4	7	3	4	1
Employee 4	3	5	4	3	6	3	8
Employee 5	3	4	4	6	3	4	4
Employee 6	3	7	4	8	3	8	4
Employee 7	6	3	5	9	2	7	9

8.5 (*Algebra: add two matrices*) Write a method to add two matrices. The header of the method is as follows:

```java
public static double[][] addMatrix(double[][] a, double[][] b)
```

In order to be added, the two matrices must have the same dimensions and the same or compatible types of elements. Let **c** be the resulting matrix. Each element c_{ij} is $a_{ij} + b_{ij}$. For example, for two 3 × 3 matrices **a** and **b**, **c** is

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{pmatrix}$$

Write a test program that prompts the user to enter two 3 × 3 matrices and displays their sum. Here is a sample run:

```
Enter matrix1: 1 2 3 4 5 6 7 8 9 ↵Enter
Enter matrix2: 0 2 4 1 4.5 2.2 1.1 4.3 5.2 ↵Enter
The matrices are added as follows
 1.0 2.0 3.0      0.0 2.0 4.0      1.0 4.0 7.0
 4.0 5.0 6.0  +   1.0 4.5 2.2  =   5.0 9.5 8.2
 7.0 8.0 9.0      1.1 4.3 5.2      8.1 12.3 14.2
```

****8.6** (*Algebra: multiply two matrices*) Write a method to multiply two matrices. The header of the method is:

VideoNote
Multiply two matrices

```java
public static double[][]
    multiplyMatrix(double[][] a, double[][] b)
```

To multiply matrix **a** by matrix **b**, the number of columns in **a** must be the same as the number of rows in **b**, and the two matrices must have elements of the same or compatible types. Let **c** be the result of the multiplication. Assume the column size of matrix a is **n**. Each element c_{ij} is $a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + \cdots + a_{in} \times b_{nj}$. For example, for two 3 × 3 matrices **a** and **b**, **c** is

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

where $c_{ij} = a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + a_{i3} \times b_{3j}$.

Write a test program that prompts the user to enter two 3 × 3 matrices and displays their product. Here is a sample run:

```
Enter matrix1: 1 2 3 4 5 6 7 8 9 ↵Enter
Enter matrix2: 0 2 4 1 4.5 2.2 1.1 4.3 5.2 ↵Enter
The multiplication of the matrices is
 1 2 3      0 2.0 4.0      5.3 23.9 24
 4 5 6   *  1 4.5 2.2  =   11.6 56.3 58.2
 7 8 9      1.1 4.3 5.2    17.9 88.7 92.4
```

***8.7** (*Points nearest to each other*) Listing 8.3 gives a program that finds two points in a two-dimensional space nearest to each other. Revise the program so it finds two points in a three-dimensional space nearest to each other. Use a two-dimensional array to represent the points. Test the program using the following points:

```
double[][] points = {{-1, 0, 3}, {-1, -1, -1}, {4, 1, 1},
  {2, 0.5, 9}, {3.5, 2, -1}, {3, 1.5, 3}, {-1.5, 4, 2},
  {5.5, 4, -0.5}};
```

The formula for computing the distance between two points `(x1, y1, z1)` and `(x2, y2, z2)` is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$.

****8.8** (*All closest pairs of points*) Revise Listing 8.3, FindNearestPoints.java, to display all closest pairs of points with the same minimum distance. Here is a sample run:

```
Enter the number of points: 8 ↵Enter
Enter 8 points: 0 0 1 1 -1 -1 2 2 -2 -2 -3 -3 -4 -4 5 5 ↵Enter
The closest two points are (0.0, 0.0) and (1.0, 1.0)
The closest two points are (0.0, 0.0) and (-1.0, -1.0)
The closest two points are (1.0, 1.0) and (2.0, 2.0)
The closest two points are (-1.0, -1.0) and (-2.0, -2.0)
The closest two points are (-2.0, -2.0) and (-3.0, -3.0)
The closest two points are (-3.0, -3.0) and (-4.0, -4.0)
Their distance is 1.4142135623730951
```

*****8.9** (*Game: play a tic-tac-toe game*) In a game of tic-tac-toe, two players take turns marking an available cell in a 3 × 3 grid with their respective tokens (either X or O). When one player has placed three tokens in a horizontal, vertical, or diagonal row on the grid, the game is over and that player has won. A draw (no winner) occurs when all the cells on the grid have been filled with tokens and neither player has achieved a win. Create a program for playing a tic-tac-toe game.

The program prompts two players to alternately enter an X token and O token. Whenever a token is entered, the program redisplays the board on the console and determines the status of the game (win, draw, or continue). Here is a sample run:

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Enter a row (0, 1, or 2) for player X: 1 ↵Enter
Enter a column (0, 1, or 2) for player X: 1 ↵Enter

-------------
|   |   |   |
-------------
|   | X |   |
-------------
|   |   |   |
-------------
Enter a row (0, 1, or 2) for player O: 1 ↵Enter
Enter a column (0, 1, or 2) for player O: 2 ↵Enter

-------------
|   |   |   |
-------------
|   | X | O |
-------------
|   |   |   |
-------------
```

```
Enter a row (0, 1, or 2) for player X:

. . .

-------------
| X |   |   |
-------------
| O | X | O |
-------------
|   |   | X |
-------------
X player won
```

*8.10 (*Largest row and column*) Write a program that randomly fills in 0s and 1s into a 4-by-4 matrix, prints the matrix, and finds the first row and column with the most 1s. Here is a sample run of the program:

```
0011
0011
1101
1010
The largest row index: 2
The largest column index: 2
```

**8.11 (*Game: nine heads and tails*) Nine coins are placed in a 3-by-3 matrix with some face up and some face down. You can represent the state of the coins using a 3-by-3 matrix with values `0` (heads) and `1` (tails). Here are some examples:

```
0 0 0    1 0 1    1 1 0    1 0 1    1 0 0
0 1 0    0 0 1    1 0 0    1 1 0    1 1 1
0 0 0    1 0 0    0 0 1    1 0 0    1 1 0
```

Each state can also be represented using a binary number. For example, the preceding matrices correspond to the numbers

```
000010000 101001100 110100001 101110100 100111110
```

There are a total of 512 possibilities, so you can use decimal numbers 0, 1, 2, 3, . . . , and 511 to represent all states of the matrix. Write a program that prompts the user to enter a number between 0 and 511 and displays the corresponding matrix with the characters `H` and `T`. In the following sample run, the user entered `7`, which corresponds to `000000111`. Since `0` stands for `H` and 1 for `T`, the output is correct.

```
Enter a number between 0 and 511: 7 Enter
H H H
H H H
T T T
```

**8.12 (*Financial application: compute tax*) Rewrite Listing 3.5, ComputeTax.java, using arrays. For each filing status, there are six tax rates. Each rate is applied to a certain amount of taxable income. For example, from the taxable income of $400,000 for a single filer, $8,350 is taxed at 10%, (33,950 − 8,350) at 15%, (82,250 − 33,950) at 25%, (171,550 − 82,550) at 28%, (372,550 − 82,250) at 33%, and (400,000 − 372,950) at 36%. The six rates are the same for all filing statuses, which can be represented in the following array:

```
double[] rates = {0.10, 0.15, 0.25, 0.28, 0.33, 0.35};
```

The brackets for each rate for all the filing statuses can be represented in a two-dimensional array as follows:

```
int[][] brackets = {
  {8350, 33950, 82250, 171550, 372950},   // Single filer
  {16700, 67900, 137050, 20885, 372950}, // Married jointly
                                          // -or qualifying widow(er)
  {8350, 33950, 68525, 104425, 186475},   // Married separately
  {11950, 45500, 117450, 190200, 372950} // Head of household
};
```

Suppose the taxable income is $400,000 for single filers. The tax can be computed as follows:

```
tax = brackets[0][0] * rates[0] +
  (brackets[0][1] - brackets[0][0]) * rates[1] +
  (brackets[0][2] - brackets[0][1]) * rates[2] +
  (brackets[0][3] - brackets[0][2]) * rates[3] +
  (brackets[0][4] - brackets[0][3]) * rates[4] +
  (400000 - brackets[0][4]) * rates[5];
```

***8.13** (*Locate the largest element*) Write the following method that returns the location of the largest element in a two-dimensional array:

```
public static int[] locateLargest(double[][] a)
```

The return value is a one-dimensional array that contains two elements. These two elements indicate the row and column indices of the largest element in the two-dimensional array. If there are more than one largest element, return the smallest row index and then the smallest column index.

Write a test program that prompts the user to enter a two-dimensional array and displays the location of the largest element in the array. Here is a sample run:

```
Enter the number of rows and columns of the array: 3 4 ↵Enter
Enter the array:
23.5 35 2 10 ↵Enter
4.5 3 45 3.5 ↵Enter
35 44 5.5 9.6 ↵Enter
The location of the largest element is at (1, 2)
```

****8.14** (*Explore matrix*) Write a program that prompts the user to enter the length of a square matrix, randomly fills in `0`s and `1`s into the matrix, prints the matrix, and finds the rows, columns, and diagonals with all `0`s or `1`s. Here is a sample run of the program:

```
Enter the size for the matrix: 4 ↵Enter
0111
0000
0100
1111
All 0s on row 2
All 1s on row 4
No same numbers on a column
No same numbers on the major diagonal
No same numbers on the sub-diagonal
```

***8.15** (*Geometry: same line?*) Programming Exercise 6.39 gives a method for testing whether three points are on the same line.

Write the following method to test whether all the points in the array `points` are on the same line:

```
public static boolean sameLine(double[][] points)
```

Write a program that prompts the user to enter five points and displays whether they are on the same line. Here are sample runs:

```
Enter five points: 3.4 2 6.5 9.5 2.3 2.3 5.5 5 -5 4 ↵Enter
The five points are not on the same line
```

```
Enter five points: 1 1 2 2 3 3 4 4 5 5 ↵Enter
The five points are on the same line
```

***8.16** (*Sort two-dimensional array*) Write a method to sort a two-dimensional array using the following header:

```
public static void sort(int m[][])
```

The method performs a primary sort on rows, and a secondary sort on columns. For example, the following array

```
{{4, 2},{1, 7},{4, 5},{1, 2},{1, 1},{4, 1}}
```

will be sorted to

```
{{1, 1},{1, 2},{1, 7},{4, 1},{4, 2},{4, 5}}.
```

*****8.17** (*Financial tsunami*) Banks lend money to each other. In tough economic times, if a bank goes bankrupt, it may not be able to pay back the loan. A bank's total assets are its current balance plus its loans to other banks. The diagram in Figure 8.8 shows five banks. The banks' current balances are 25, 125, 175, 75, and 181 million dollars, respectively. The directed edge from node 1 to node 2 indicates that bank 1 lends 40 million dollars to bank 2.

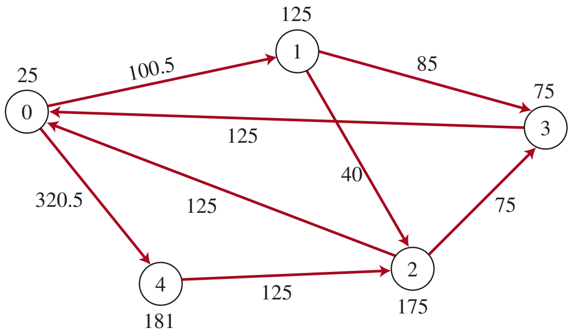


FIGURE 8.8 Banks lend money to each other.

If a bank's total assets are under a certain limit, the bank is unsafe. The money it borrowed cannot be returned to the lender, and the lender cannot count the loan in its total assets. Consequently, the lender may also be unsafe, if its total assets are under the limit. Write a program to find all the unsafe banks. Your program reads the input as follows. It first reads two integers `n` and `limit`, where `n` indicates the number of banks and `limit` is the minimum total assets for keeping a bank safe. It then reads `n` lines that describe the information for `n` banks with IDs from `0` to `n-1`.

The first number in the line is the bank's balance, the second number indicates the number of banks that borrowed money from the bank, and the rest are pairs of two numbers. Each pair describes a borrower. The first number in the pair is the borrower's ID and the second is the amount borrowed. For example, the input for the five banks in Figure 8.8 is as follows (note the limit is 201):

```
5 201
25 2 1 100.5 4 320.5
125 2 2 40 3 85
175 2 0 125 3 75
75 1 0 125
181 1 2 125
```

The total assets of bank 3 are (75 + 125), which is under 201, so bank 3 is unsafe. After bank 3 becomes unsafe, the total assets of bank 1 fall below (125 + 40). Thus, bank 1 is also unsafe. The output of the program should be

```
Unsafe banks are 3 1
```

(*Hint*: Use a two-dimensional array `borrowers` to represent loans. `borrowers[i][j]` indicates the loan that bank `i` provides to bank `j`. Once bank `j` becomes unsafe, `borrowers[i][j]` should be set to `0`.)

***8.18** (*Shuffle rows*) Write a method that shuffles the rows in a two-dimensional `int` array using the following header:

```
public static void shuffle(int[][] m)
```

Write a test program that shuffles the following matrix:

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}};
```

****8.19** (*Pattern recognition: four consecutive equal numbers*) Write the following method that tests whether a two-dimensional array has four consecutive numbers of the same value, either horizontally, vertically, or diagonally:

```
public static boolean isConsecutiveFour(int[][] values)
```

Write a test program that prompts the user to enter the number of rows and columns of a two-dimensional array then the values in the array, and displays true if the array contains four consecutive numbers with the same value. Otherwise, the program displays false. Here are some examples of the true cases:

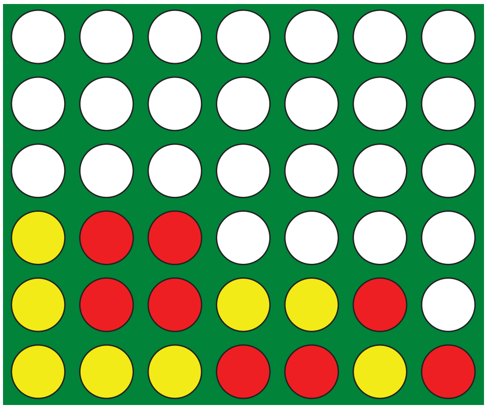
0	1	0	3	1	6	1
0	1	6	8	6	0	1
5	6	2	1	8	2	9
6	5	6	1	1	9	1
1	3	6	1	4	0	7
3	3	3	3	4	0	7

0	1	0	3	1	6	1
0	1	6	8	6	0	1
5	5	2	1	8	2	9
6	5	6	1	1	9	1
1	5	6	1	4	0	7
3	5	3	3	4	0	7

0	1	0	3	1	6	1
0	1	6	8	6	0	1
5	6	2	1	6	2	9
6	5	6	6	1	9	1
1	3	6	1	4	0	7
3	6	3	3	4	0	7

0	1	0	3	1	6	1
0	1	6	8	6	0	1
9	6	2	1	8	2	9
6	9	6	1	1	9	1
1	3	9	1	4	0	7
3	3	3	9	4	0	7

*****8.20** (*Game: connect four*) Connect four is a two-player board game in which the players alternately drop colored disks into a seven-column, six-row vertically suspended grid, as shown below.



The objective of the game is to connect four same-colored disks in a row, a column, or a diagonal before your opponent can do likewise. The program prompts two players to drop a red or yellow disk alternately. Whenever a disk is dropped, the program redisplays the board on the console and determines the status of the game (win, draw, or continue). Here is a sample run:

```
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
---------------
Drop a red disk at column (0–6): 0

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|R| | | | | | |
---------------
```

```
Drop a yellow disk at column (0–6): 3 ↵Enter

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|R| | |Y| | | |

 . . .
 . . .
 . . .

Drop a yellow disk at column (0–6): 6 ↵Enter
| | | | | | | |
| | | | | | | |
| | | |R| | | |
| | | |Y|R|Y| |
| | |R|Y|Y|Y|Y|
|R|Y|R|Y|R|R|R|
-----------------------------
The yellow player won
```

***8.21** (*Central city*) Given a set of cities, the central city is the city that has the shortest total distance to all other cities. Write a program that prompts the user to enter the number of cities and the locations of the cities (coordinates), and finds the central city and its total distance to all other cities.

```
Enter the number of cities: 5 ↵Enter
Enter the coordinates of the cities:
  2.5 5 5.1 3 1 9 5.4 54 5.5 2.1 ↵Enter
The central city is at (2.5, 5.0)
The total distance to all other cities is 60.81
```

VideoNote
Even number of 1s

***8.22** (*Even number of 1s*) Write a program that generates a 6-by-6 two-dimensional matrix filled with 0s and 1s, displays the matrix, and checks if every row and every column have an even number of 1s.

***8.23** (*Game: find the flipped cell*) Suppose you are given a 6-by-6 matrix filled with 0s and 1s. All rows and all columns have an even number of 1s. Let the user flip one cell (i.e., flip from 1 to 0 or from 0 to 1) and write a program to find which cell was flipped. Your program should prompt the user to enter a 6-by-6 array with 0s and 1s and find the first row *r* and first column *c* where the even number of the 1s property is violated (i.e., the number of 1s is not even). The flipped cell is at (*r*, *c*). Here is a sample run:

```
Enter a 6-by-6 matrix row by row:
1 1 1 0 1 1 ↵Enter
1 1 1 1 0 0 ↵Enter
0 1 0 1 1 1 ↵Enter
1 1 1 1 1 1 ↵Enter
0 1 1 1 1 0 ↵Enter
1 0 0 0 0 1 ↵Enter
The flipped cell is at (0, 1)
```

***8.24** (*Check Sudoku solution*) Listing 8.4 checks whether a solution is valid by checking whether every number is valid in the board. Rewrite the program by checking whether every row, every column, and every small box has the numbers 1 to 9.

***8.25** (*Markov matrix*) An $n \times n$ matrix is called a *positive Markov matrix* if each element is positive and the sum of the elements in each column is 1. Write the following method to check whether a matrix is a Markov matrix:

```java
public static boolean isMarkovMatrix(double[][] m)
```

Write a test program that prompts the user to enter a 3×3 matrix of double values and tests whether it is a Markov matrix. Here are sample runs:

```
Enter a 3-by-3 matrix row by row:
0.15 0.875 0.375 ↵Enter
0.55 0.005 0.225 ↵Enter
0.30 0.12 0.4 ↵Enter
It is a Markov matrix
```

```
Enter a 3-by-3 matrix row by row:
0.95 -0.875 0.375 ↵Enter
0.65 0.005 0.225 ↵Enter
0.30 0.22 -0.4 ↵Enter
It is not a Markov matrix
```

***8.26** (*Row sorting*) Implement the following method to sort the rows in a two-dimensional array. A new array is returned and the original array is intact.

```java
public static double[][] sortRows(double[][] m)
```

Write a test program that prompts the user to enter a 3×3 matrix of double values and displays a new row-sorted matrix. Here is a sample run:

```
Enter a 3-by-3 matrix row by row:
0.15 0.875 0.375 ↵Enter
0.55 0.005 0.225 ↵Enter
0.30 0.12 0.4 ↵Enter

The row-sorted array is
0.15 0.375 0.875
0.005 0.225 0.55
0.12 0.30 0.4
```

***8.27** (*Column sorting*) Implement the following method to sort the columns in a two-dimensional array. A new array is returned and the original array is intact.

```java
public static double[][] sortColumns(double[][] m)
```

Write a test program that prompts the user to enter a 3 × 3 matrix of double values and displays a new column-sorted matrix. Here is a sample run:

```
Enter a 3-by-3 matrix row by row:
0.15 0.875 0.375 ↵Enter
0.55 0.005 0.225 ↵Enter
0.30 0.12 0.4 ↵Enter

The column-sorted array is
0.15 0.0050 0.225
0.3  0.12   0.375
0.55 0.875  0.4
```

8.28 (*Strictly identical arrays*) The two-dimensional arrays `m1` and `m2` are *strictly identical* if their corresponding elements are equal. Write a method that returns `true` if `m1` and `m2` are strictly identical, using the following header:

```
public static boolean equals(int[][] m1, int[][] m2)
```

Write a test program that prompts the user to enter two 3 × 3 arrays of integers and displays whether the two are strictly identical. Here are the sample runs:

```
Enter list1: 51 22 25 6 1 4 24 54 6 ↵Enter
Enter list2: 51 22 25 6 1 4 24 54 6 ↵Enter
The two arrays are strictly identical
```

```
Enter list1: 51 25 22 6 1 4 24 54 6 ↵Enter
Enter list2: 51 22 25 6 1 4 24 54 6 ↵Enter
The two arrays are not strictly identical
```

8.29 (*Identical arrays*) The two-dimensional arrays `m1` and `m2` are *identical* if they have the same contents. Write a method that returns `true` if `m1` and `m2` are identical, using the following header:

```
public static boolean equals(int[][] m1, int[][] m2)
```

Write a test program that prompts the user to enter two 3 × 3 arrays of integers and displays whether the two are identical. Here are the sample runs:

```
Enter list1: 51 25 22 6 1 4 24 54 6 ↵Enter
Enter list2: 51 22 25 6 1 4 24 54 6 ↵Enter
The two arrays are identical
```

```
Enter list1: 51 5 22 6 1 4 24 54 6 ↵Enter
Enter list2: 51 22 25 6 1 4 24 54 6 ↵Enter
The two arrays are not identical
```

***8.30** (*Algebra: solve linear equations*) Write a method that solves the following 2 × 2 system of linear equations:

$$\begin{aligned} a_{00}x + a_{01}y &= b_0 \\ a_{10}x + a_{11}y &= b_1 \end{aligned} \qquad x = \frac{b_0 a_{11} - b_1 a_{01}}{a_{00}a_{11} - a_{01}a_{10}} \qquad y = \frac{b_1 a_{00} - b_0 a_{10}}{a_{00}a_{11} - a_{01}a_{10}}$$

The method header is:

```java
public static double[] linearEquation(double[][] a, double[] b)
```

The method returns `null` if $a_{00}a_{11} - a_{01}a_{10}$ is `0`. Write a test program that prompts the user to enter a_{00}, a_{01}, a_{10}, a_{11}, b_0, and b_1 and displays the result. If $a_{00}a_{11} - a_{01}a_{10}$ is `0`, report that "The equation has no solution." A sample run is similar to Programming Exercise 3.3.

***8.31** (*Geometry: intersecting point*) Write a method that returns the intersecting point of two lines. The intersecting point of the two lines can be found by using the formula given in Programming Exercise 3.25. Assume that (`x1`, `y1`) and (`x2`, `y2`) are the two points on line 1 and (`x3`, `y3`) and (`x4`, `y4`) are on line 2. The method header is:

```java
public static double[] getIntersectingPoint(double[][] points)
```

The points are stored in a 4-by-2 two-dimensional array `points` with (`points[0][0]`, `points[0][1]`) for (`x1`, `y1`). The method returns the intersecting point or null if the two lines are parallel. Write a program that prompts the user to enter four points and displays the intersecting point. See Programming Exercise 3.25 for a sample run.

***8.32** (*Geometry: area of a triangle*) Write a method that returns the area of a triangle using the following header:

```java
public static double getTriangleArea(double[][] points)
```

The points are stored in a 3-by-2 two-dimensional array `points` with `points[0][0]` and `points[0][1]` for (`x1`, `y1`). The triangle area can be computed using the formula in Programming Exercise 2.19. The method returns `0` if the three points are on the same line. Write a program that prompts the user to enter three points of a triangle and displays the triangle's area. Here are the sample runs:

```
Enter x1, y1, x2, y2, x3, y3: 2.5 2 5 -1.0 4.0 2.0 ↵Enter
The area of the triangle is 2.25
```

```
Enter x1, y1, x2, y2, x3, y3: 2 2 4.5 4.5 6 6 ↵Enter
The three points are on the same line
```

***8.33** (*Geometry: polygon subareas*) A convex four-vertex polygon is divided into four triangles, as shown in Figure 8.9.

Write a program that prompts the user to enter the coordinates of four vertices and displays the areas of the four triangles in increasing order. Here is a sample run:

```
Enter x1, y1, x2, y2, x3, y3, x4, y4:
  -2.5 2 4 4 3 -2 -2 -3.5 ↵Enter
The areas are 6.17 7.96 8.08 10.42
```

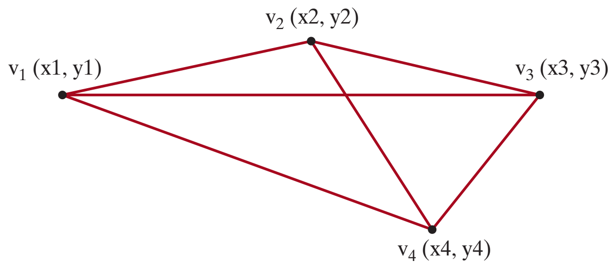


FIGURE 8.9 A four-vertex polygon is defined by four vertices.

***8.34** (*Geometry: rightmost lowest point*) In computational geometry, often you need to find the rightmost lowest point in a set of points. Write the following method that returns the rightmost lowest point in a set of points:

```
public static double[]
  getRightmostLowestPoint(double[][] points)
```

Write a test program that prompts the user to enter the coordinates of six points and displays the rightmost lowest point. Here is a sample run:

```
Enter 6 points: 1.5 2.5 -3 4.5 5.6 -7 6.5 -7 8 1 10 2.5 ↵Enter
The rightmost lowest point is (6.5, -7.0)
```

****8.35** (*Largest block*) Given a square matrix with the elements 0 or 1, write a program to find a maximum square submatrix whose elements are all 1s. Your program should prompt the user to enter the number of rows in the matrix. The program then displays the location of the first element in the maximum square submatrix and the number of rows in the submatrix. Here is a sample run:

```
Enter the number of rows in the matrix: 5 ↵Enter
Enter the matrix row by row:
1 0 1 0 1 ↵Enter
1 1 1 0 1 ↵Enter
1 0 1 1 1 ↵Enter
1 0 1 1 1 ↵Enter
1 0 1 1 1 ↵Enter

The maximum square submatrix is at (2, 2) with size 3
```

Your program should implement and use the following method to find the maximum square submatrix:

```
public static int[] findLargestBlock(int[][] m)
```

The return value is an array that consists of three values. The first two values are the row and column indices for the first element in the submatrix, and the third value is the number of the rows in the submatrix. For an animation of this problem, see https://liveexample.pearsoncmg.com/dsanimation/LargestBlockeBook.html.

****8.36** (*Latin square*) A Latin square is an *n*-by-*n* array filled with `n` different Latin letters, each occurring exactly once in each row and once in each column. Write a program that prompts the user to enter the number `n` and the array of characters, as shown in the sample output, and checks if the input array is a Latin square. The characters are the first `n` characters starting from `A`.

```
Enter number n: 4 ↵Enter
Enter 4 rows of letters separated by spaces:
A B C D ↵Enter
B A D C ↵Enter
C D B A ↵Enter
D C A B ↵Enter
The input array is a Latin square
```

```
Enter number n: 3 ↵Enter
Enter 3 rows of letters separated by spaces:
A F D ↵Enter
Wrong input: the letters must be from A to C
```

****8.37** (*Guess the capitals*) Write a program that repeatedly prompts the user to enter a capital for a state. Upon receiving the user input, the program reports whether the answer is correct. Assume that `50` states and their capitals are stored in a two-dimensional array, as shown in Figure 8.10. The program prompts the user to answer all states' capitals and displays the total correct count. The user's answer is not case-sensitive.

Alabama	Montgomery
Alaska	Juneau
Arizona	Phoenix
...	...
...	...

FIGURE 8.10 A two-dimensional array stores states and their capitals.

Here is a sample run:

```
What is the capital of Alabama? Montogomery ↵Enter
The correct answer should be Montgomery
What is the capital of Alaska? Juneau ↵Enter
Your answer is correct
What is the capital of Arizona? ...
...
The correct count is 35
```

CHAPTER

9

OBJECTS AND CLASSES

Objectives

- To describe objects and classes, and use classes to model objects (§9.2).
- To use UML graphical notation to describe classes and objects (§9.2).
- To demonstrate how to define classes and create objects (§9.3).
- To create objects using constructors (§9.4).
- To define a reference variable using a reference type and access objects via object reference variables (§9.5).
- To access an object's data and methods using the object member access operator (`.`) (§9.5.1).
- To define data fields of reference types and assign default values for an object's data fields (§9.5.2).
- To distinguish between object reference variables and primitive-data-type variables (§9.5.3).
- To use the Java library classes `Date`, `Random`, and `Point2D` (§9.6).
- To distinguish between instance and static variables and methods (§9.7).
- To define private data fields with appropriate getter and setter methods (§9.8).
- To encapsulate data fields to make classes easy to maintain (§9.9).
- To develop methods with object arguments and differentiate between primitive-type arguments and object-type arguments (§9.10).
- To store and process objects in arrays (§9.11).
- To create immutable objects from immutable classes to protect the contents of objects (§9.12).
- To determine the scope of variables in the context of a class (§9.13).
- To use the keyword `this` to refer to the calling object itself (§9.14).

9.1 Introduction

Key Point

Object-oriented programming enables you to develop large-scale software and GUIs effectively.

why OOP?

Object-oriented programming is essentially a technology for developing reusable software. Having learned the material in the preceding chapters, you are able to solve many programming problems using selections, loops, methods, and arrays. However, these Java features are not sufficient for developing graphical user interfaces and large-scale software systems. Suppose you want to develop a graphical user interface (GUI, pronounced *goo-ee*) as shown in Figure 9.1. How would you program it?

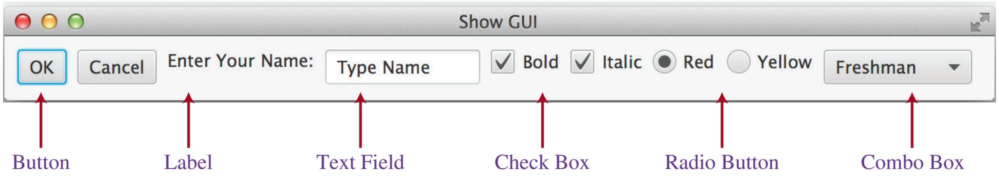


FIGURE 9.1 The GUI objects are created from classes.

This chapter introduces object-oriented programming, which you can use to develop GUI and large-scale software systems.

9.2 Defining Classes for Objects

Key Point

A class defines the properties and behaviors for objects.

VideoNote
Define classes and create objects

object
state of an object
properties
attributes
data fields
behavior
actions

Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behavior.

- The *state* of an object (also known as its *properties* or *attributes*) is represented by *data fields* with their current values. A circle object, for example, has a data field `radius`, which is the property that characterizes a circle. A rectangle object, for example, has the data fields `width` and `height`, which are the properties that characterize a rectangle.
- The *behavior* of an object (also known as its *actions*) is defined by methods. To invoke a method on an object is to ask the object to perform an action. For example, you may define methods named `getArea()` and `getPerimeter()` for circle objects. A circle object may invoke `getArea()` to return its area and `getPerimeter()` to return its perimeter. You may also define the `setRadius(radius)` method. A circle object can invoke this method to change its radius.

class
contract
instantiation
instance

Objects of the same type are defined using a common class. A *class* is a template, blueprint, or *contract* that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class. Creating an instance is referred to as *instantiation*. The terms *object* and *instance* are often interchangeable. The relationship between classes and objects is analogous to that between an apple-pie recipe and apple pies: You can make as many apple pies as you want from a single recipe. Figure 9.2 shows a class named `Circle` and its three objects.

data field
method
constructors

A Java class uses variables to define data fields and methods to define actions. In addition, a class provides methods of a special type, known as *constructors*, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects. Figure 9.3 shows an example of defining the class for circle objects.

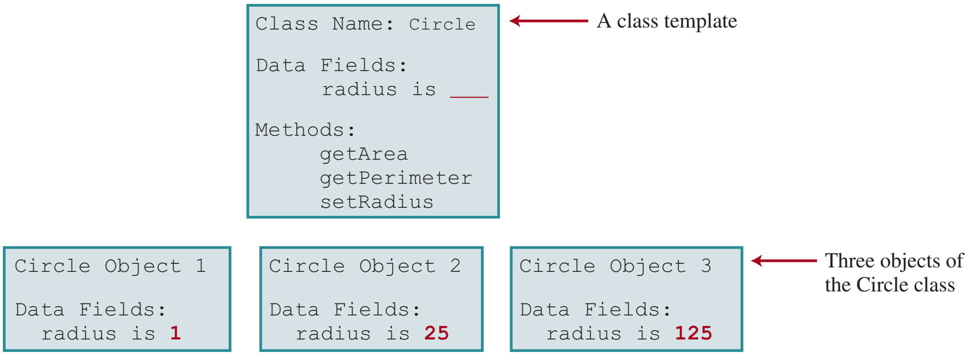
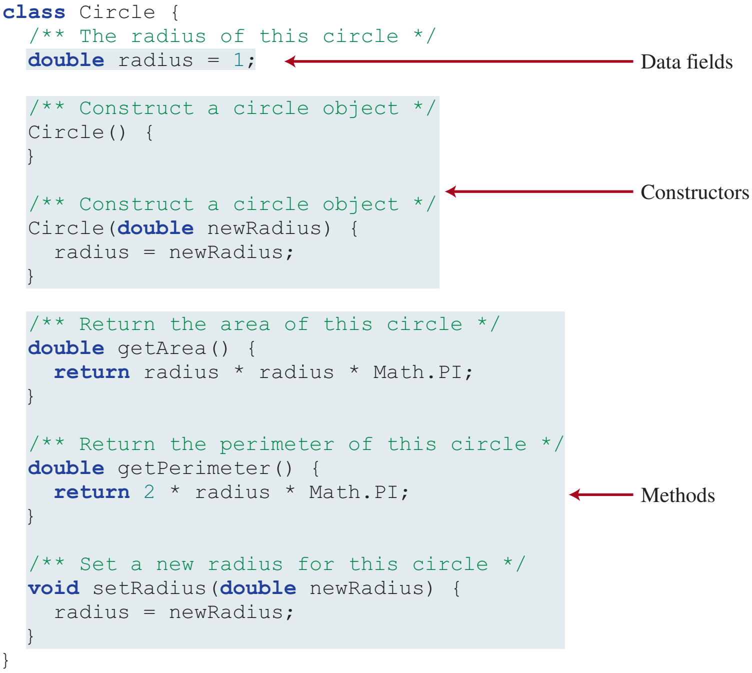

Class Name: Circle
Data Fields:
radius is ___
Methods:
getArea
getPerimeter
setRadius
A class template
Circle Object 1
Data Fields:
radius is 1
Circle Object 2
Data Fields:
radius is 25
Circle Object 3
Data Fields:
radius is 125
Three objects of the Circle class

FIGURE 9.2 A class is a template for creating objects.



```
class Circle {
  /** The radius of this circle */
  double radius = 1;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }

  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set a new radius for this circle */
  void setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

Data fields
Constructors
Methods

FIGURE 9.3 A class is a construct that defines objects of the same type.

The `Circle` class is different from all of the other classes you have seen thus far. It does not have a `main` method, and therefore cannot be run; it is merely a definition for circle objects. The class that contains the `main` method will be referred to in this book, for convenience, as the *main class*.

main class

The illustration of class templates and objects in Figure 9.2 can be standardized using *Unified Modeling Language (UML)* notation. This notation, as shown in Figure 9.4, is called a *UML class diagram*, or simply a *class diagram*. In the class diagram, the data field is denoted as

Unified Modeling Language (UML)

class diagram

```
dataFieldName: dataFieldType
```

The constructor is denoted as

```
ClassName(parameterName: parameterType)
```

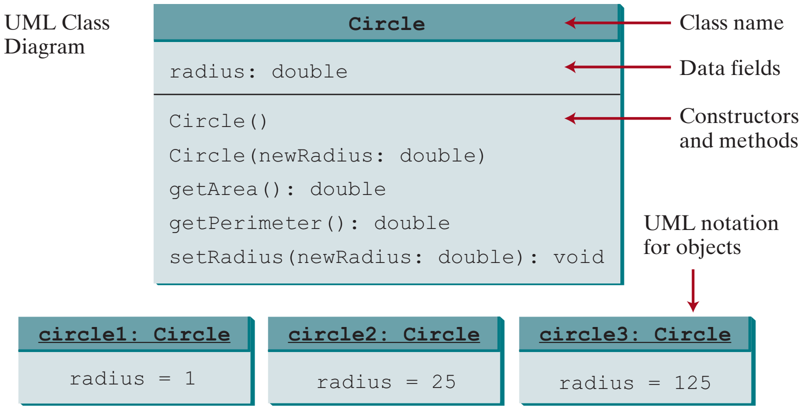

UML Class Diagram
Circle
Class name
radius: double
Data fields
Circle()
Circle(newRadius: double)
getArea(): double
getPerimeter(): double
setRadius(newRadius: double): void
Constructors and methods
UML notation for objects
circle1: Circle
radius = 1
circle2: Circle
radius = 25
circle3: Circle
radius = 125

FIGURE 9.4 Classes and objects can be represented using UML notation.

The method is denoted as

```
methodName(parameterName: parameterType): returnType
```

9.3 Example: Defining Classes and Creating Objects

Key Point

Classes are definitions for objects and objects are created from classes.

This section gives two examples of defining classes and uses the classes to create objects. Listing 9.1 is a program that defines the **`Circle`** class and uses it to create objects. The program constructs three circle objects with radius **`1`**, **`25`**, and **`125`** and displays the **`radius`** and **`area`** of each of the three circles. It then changes the radius of the second object to **`100`** and displays its new radius and area.

LISTING 9.1 TestCircle.java

main class

`main` method

create object

create object

create object

```
public class TestCircle {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 1
    Circle circle1 = new Circle();
    System.out.println("The area of the circle of radius "
      + circle1.radius + " is " + circle1.getArea());

    // Create a circle with radius 25
    Circle circle2 = new Circle(25);
    System.out.println("The area of the circle of radius "
      + circle2.radius + "is" + circle2.getArea());

    // Create a circle with radius 125
    Circle circle3 = new Circle(125);
    System.out.println("The area of the circle of radius "
      + circle3.radius + " is " + circle3.getArea());

    // Modify circle radius
    circle2.radius = 100; // or circle2.setRadius(100)
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());
  }
}

```

```
// Define the circle class with two constructors
class Circle {
  double radius;

  /** Construct a circle with radius 1 */
  Circle() {
    radius = 1;
  }

  /** Construct a circle with a specified radius */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }

  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set a new radius for this circle */
  void setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

class `Circle`
data field
no-arg constructor
second constructor
`getArea`
`getPerimeter`
`setRadius`

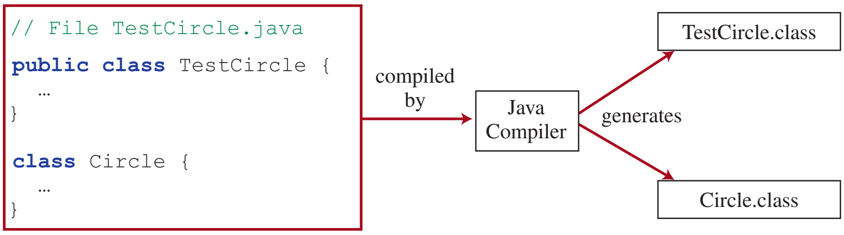
```
The area of the circle of radius 1.0 is 3.141592653589793
The area of the circle of radius 25.0 is 1963.4954084936207
The area of the circle of radius 125.0 is 49087.385212340516
The area of the circle of radius 100.0 is 31415.926535897932
```

The program contains two classes. The first of these, **`TestCircle`**, is the main class. Its sole purpose is to test the second class, **`Circle`**. Such a program that uses the class is often referred to as a *client* of the class. When you run the program, the Java runtime system invokes the **`main`** method in the main class.

client

You can put the two classes into one file, but only one class in the file can be a *public class*. Furthermore, the public class must have the same name as the file name. Therefore, the file name is **TestCircle.java**, since **`TestCircle`** is public. Each class in the source code is compiled into a **.class** file. When you compile **TestCircle.java**, two class files **TestCircle.class** and **Circle.class** are generated, as shown in Figure 9.5.

public class



```
// File TestCircle.java
public class TestCircle {
  …
}

class Circle {
  …
}
```

Figure 9.5 Each class in the source code file is compiled into a **.class** file.

The main class contains the **`main`** method (line 3) that creates three objects. As in creating an array, the **`new`** operator is used to create an object from the constructor: **`new Circle()`** creates an object with radius **`1`** (line 5), **`new Circle(25)`** creates an object with radius **`25`** (line 10), and **`new Circle(125)`** creates an object with radius **`125`** (line 15).

These three objects (referenced by **`circle1`**, **`circle2`**, and **`circle3`**) have different data but the same methods. Therefore, you can compute their respective areas by using the **`getArea()`** method. The data fields can be accessed via the reference of the object using **`circle1.radius`**, **`circle2.radius`**, and **`circle3.radius`**, respectively. The object can invoke its method via the reference of the object using **`circle1.getArea()`**, **`circle2.getArea()`**, and **`circle3.getArea()`**, respectively.

These three objects are independent. The radius of **`circle2`** is changed to **`100`** in line 20. The object's new radius and area are displayed in lines 21 and 22.

There are many ways to write Java programs. For instance, you can combine the two classes in the preceding example into one, as given in Listing 9.2.

Listing 9.2 `Circle.java (AlternativeCircle.java)`

main method

data field

no-arg constructor

second constructor

method

```
 1  public class Circle {
 2    /** Main method */
 3    public static void main(String[] args) {
 4      // Create a circle with radius 1
 5      Circle circle1 = new Circle();
 6      System.out.println("The area of the circle of radius "
 7        + circle1.radius + " is " + circle1.getArea());
 8
 9      // Create a circle with radius 25
10      Circle circle2 = new Circle(25);
11      System.out.println("The area of the circle of radius "
12        + circle2.radius + " is " + circle2.getArea());
13
14      // Create a circle with radius 125
15      Circle circle3 = new Circle(125);
16      System.out.println("The area of the circle of radius "
17        + circle3.radius + " is " + circle3.getArea());
18
19      // Modify circle radius
20      circle2.radius = 100;
21      System.out.println("The area of the circle of radius "
22        + circle2.radius + " is " + circle2.getArea());
23    }
24
25    double radius;
26
27    /** Construct a circle with radius 1 */
28    Circle() {
29      radius = 1;
30    }
31
32    /** Construct a circle with a specified radius */
33    Circle(double newRadius) {
34      radius = newRadius;
35    }
36
37    /** Return the area of this circle */
38    double getArea() {
39      return radius * radius * Math.PI;
40    }
41
```

```
  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set a new radius for this circle */
  void setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

Since the combined class has a `main` method, it can be executed by the Java interpreter. The `main` method is the same as that in Listing 9.1. This demonstrates that you can test a class by simply adding a `main` method in the same class.

As another example, consider television sets. Each TV is an object with states (current channel, current volume level, and power on or off) and behaviors (change channels, adjust volume, and turn on/off). You can use a class to model TV sets. The UML diagram for the class is shown in Figure 9.6.

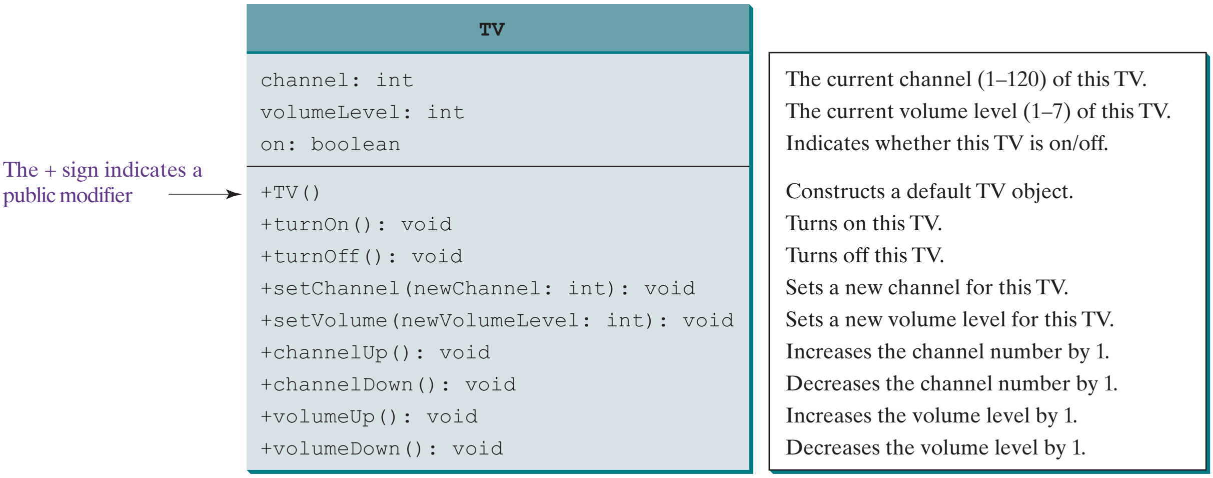

The + sign indicates a public modifier
TV
channel: int
volumeLevel: int
on: boolean
+TV()
+turnOn(): void
+turnOff(): void
+setChannel(newChannel: int): void
+setVolume(newVolumeLevel: int): void
+channelUp(): void
+channelDown(): void
+volumeUp(): void
+volumeDown(): void
The current channel (1–120) of this TV.
The current volume level (1–7) of this TV.
Indicates whether this TV is on/off.
Constructs a default TV object.
Turns on this TV.
Turns off this TV.
Sets a new channel for this TV.
Sets a new volume level for this TV.
Increases the channel number by 1.
Decreases the channel number by 1.
Increases the volume level by 1.
Decreases the volume level by 1.

FIGURE 9.6 The TV class models TV sets.

Listing 9.3 gives a program that defines the `TV` class.

LISTING 9.3 TV.java

```
public class TV {
  int channel = 1; // Default channel is 1
  int volumeLevel = 1; // Default volume level is 1
  boolean on = false; // TV is off

  public TV() {
  }

  public void turnOn() {
    on = true;
  }

  public void turnOff() {
```

data fields

constructor

turn on TV

turn off TV

set a new channel
set a new volume
increase channel
decrease channel
increase volume
decrease volume

```
    on = false;
  }

  public void setChannel(int newChannel) {
    if (on && newChannel >= 1 && newChannel <= 120)
      channel = newChannel;
  }

  public void setVolume(int newVolumeLevel) {
    if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
      volumeLevel = newVolumeLevel;
  }

  public void channelUp() {
    if (on && channel < 120)
      channel++;
  }

  public void channelDown() {
    if (on && channel > 1)
      channel--;
  }

  public void volumeUp() {
    if (on && volumeLevel < 7)
      volumeLevel++;
  }

  public void volumeDown() {
    if (on && volumeLevel > 1)
      volumeLevel--;
  }
}
```

The constructor and methods in the **TV** class are defined public so they can be accessed from other classes. Note the channel and volume level are not changed if the TV is not on. Before either of these is changed, its current value is checked to ensure it is within the correct range.

Listing 9.4 gives a program that uses the **TV** class to create two objects.

LISTING 9.4 TestTV.java

`main` method
create a TV
turn on
set a new channel
set a new volume
create a TV
turn on
increase channel
increase volume
display state

```
public class TestTV {
  public static void main(String[] args) {
    TV tv1 = new TV();
    tv1.turnOn();
    tv1.setChannel(30);
    tv1.setVolume(3);

    TV tv2 = new TV();
    tv2.turnOn();
    tv2.channelUp();
    tv2.channelUp();
    tv2.volumeUp();

    System.out.println("tv1's channel is " + tv1.channel
      + " and volume level is " + tv1.volumeLevel);
    System.out.println("tv2's channel is " + tv2.channel
      + " and volume level is " + tv2.volumeLevel);
  }
}
```

```
tv1's channel is 30 and volume level is 3
tv2's channel is 3 and volume level is 2
```

The program creates two objects in lines 3 and 8 and invokes the methods on the objects to perform actions for setting channels and volume levels and for increasing channels and volumes. The program displays the state of the objects in lines 14–17. The methods are invoked using syntax such as `tv1.turnOn()` (line 4). The data fields are accessed using syntax such as `tv1.channel` (line 14).

These examples have given you a glimpse of classes and objects. You may have many questions regarding constructors, objects, reference variables, accessing data fields, and invoking object's methods. The sections that will follow discuss these issues in detail.

Check Point

9.3.1 Describe the relationship between an object and its defining class.

9.3.2 How do you define a class?

9.3.3 How do you declare an object's reference variable?

9.3.4 How do you create an object?

9.4 Constructing Objects Using Constructors

Key Point

A constructor is invoked to create an object using the `new` *operator.*

Constructors are a special kind of method. They have three peculiarities:

- A constructor must have the same name as the class itself. constructor's name
- Constructors do not have a return type—not even `void`. no return type
- Constructors are invoked using the `new` operator when an object is created. Constructors play the role of initializing objects. new operator

overloaded constructors

The constructor has exactly the same name as its defining class. Like regular methods, constructors can be overloaded (i.e., multiple constructors can have the same name but different signatures), making it easy to construct objects with different initial data values.

It is a common mistake to put the `void` keyword in front of a constructor. For example,

no void

```
public void Circle() {
}
```

In this case, `Circle()` is a method, not a constructor.

constructing objects

Constructors are used to construct objects. To construct an object from a class, invoke a constructor of the class using the `new` operator, as follows:

```
new ClassName(arguments);
```

For example, `new Circle()` creates an object of the `Circle` class using the first constructor defined in the `Circle` class, and `new Circle(25)` creates an object using the second constructor defined in the `Circle` class.

no-arg constructor

A class normally provides a constructor without arguments (e.g., `Circle()`). Such a constructor is referred to as a *no-arg* or *no-argument constructor*.

default constructor

A class may be defined without constructors. In this case, a public no-arg constructor with an empty body is implicitly defined in the class. This constructor, called a *default constructor*, is provided automatically *only if no constructors are explicitly defined in the class*.

Check Point

9.4.1 What are the differences between constructors and methods?

9.4.2 When will a class have a default constructor?

9.4.3 What is wrong with each of the following programs?

```
public class ShowErrors {
  public static void main(String[] args) {
    ShowErrors t = new ShowErrors(5);
  }
}
```

(a)

```
public class ShowErrors {
  public static void main(String[] args) {
    C c = new C(5.0);
    System.out.println(c.value);
  }
}

class C {
  int value = 2;
}
```

(b)

9.4.4 What is wrong in the following code?

```
class Test {
  public static void main(String[] args) {
    A a = new A();
    a.print();
  }
}

class A {
  String s;

  A(String newS) {
    s = newS;
  }

  public void print() {
    System.out.print(s);
  }
}
```

9.5 Accessing Objects via Reference Variables

Key Point

An object's data and methods can be accessed through the dot (.) operator via the object's reference variable.

Newly created objects are allocated in the memory. They can be accessed via reference variables.

reference variable

Objects are accessed via the object's *reference variables*, which contain references to the objects. Such variables are declared using the following syntax:

```
ClassName objectRefVar;
```

reference type

A class is essentially a programmer-defined type. A class is a *reference type*, which means that a variable of the class type can reference an instance of the class. The following statement declares the variable `myCircle` to be of the `Circle` type:

```
Circle myCircle;
```

The variable `myCircle` can reference a `Circle` object. The next statement creates an object and assigns its reference to `myCircle`:

```
myCircle = new Circle();
```

You can write a single statement that combines the declaration of an object reference variable, the creation of an object, and the assigning of an object reference to the variable with the following syntax:

```
ClassName objectRefVar = new ClassName();
```

Here is an example:

```
Circle myCircle = new Circle();
```

The variable `myCircle` holds a reference to a `Circle` object.

object vs. object reference variable

Note

An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored. Therefore, it is fine, for simplicity, to say that `myCircle` is a `Circle` object rather than use the long-winded description that `myCircle` is a variable that contains a reference to a `Circle` object.

array object

Note

Arrays are treated as objects in Java. Arrays are created using the `new` operator. An array variable is actually a variable that contains a reference to an array.

9.5.1 Accessing an Object's Data and Methods

dot operator (.)

In OOP terminology, an object's member refers to its data fields and methods. After an object is created, its data can be accessed and its methods can be invoked using the *dot operator* (`.`), also known as the *object member access operator*:

- `objectRefVar.dataField` references a data field in the object.
- `objectRefVar.method(arguments)` invokes a method on the object.

For example, `myCircle.radius` references the radius in `myCircle` and `myCircle.getArea()` invokes the `getArea` method on `myCircle`. Methods are invoked as operations on objects.

instance variable

instance method

calling object

The data field `radius` is referred to as an *instance variable* because it is dependent on a specific instance. For the same reason, the method `getArea` is referred to as an *instance method* because you can invoke it only on a specific instance. The object on which an instance method is invoked is called a *calling object*.

invoking methods

Caution

Recall that you use `Math.methodName(arguments)` (e.g., `Math.pow(3, 2.5)`) to invoke a method in the `Math` class. Can you invoke `getArea()` using `Circle.getArea()`? The answer is no. All the methods in the `Math` class are static methods, which are defined using the `static` keyword. However, `getArea()` is an instance method, and thus nonstatic. It must be invoked from an object using `objectRefVar.methodName(arguments)` (e.g., `myCircle.getArea()`). Further explanation will be given in Section 9.7, Static Variables, Constants, and Methods.

Note

Usually you create an object and assign it to a variable, then later you can use the variable to reference the object. Occasionally, an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable using the syntax:

```
new Circle();
```

or

```
System.out.println("Area is " + new Circle(5).getArea());
```

The former statement creates a `Circle` object. The latter creates a `Circle` object and invokes its `getArea` method to return its area. An object created in this way is known as an *anonymous object*.

anonymous object

9.5.2 Reference Data Fields and the `null` Value

reference data fields

The data fields can be of reference types. For example, the following **`Student`** class contains a data field **`name`** of the **`String`** type. **`String`** is a predefined Java class.

```
class Student {
  String name; // name has the default value null
  int age; // age has the default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // gender has default value '\u0000'
}
```

null value

If a data field of a reference type does not reference any object, the data field holds a special Java value, **`null`**. **`null`** is a literal just like **`true`** and **`false`**. While **`true`** and **`false`** are Boolean literals, **`null`** is a literal for a reference type. **`null`** is not a Java keyword, but it is a reserved word in Java.

default field values

The default value of a data field is **`null`** for a reference type, **`0`** for a numeric type, **`false`** for a **`boolean`** type, and **`\u0000`** for a **`char`** type. However, Java assigns no default value to a local variable inside a method. The following code displays the default values of the data fields **`name`**, **`age`**, **`isScienceMajor`**, and **`gender`** for a **`Student`** object:

```
class TestStudent {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println("name? " + student.name);
    System.out.println("age? " + student.age);
    System.out.println("isScienceMajor? " + student.isScienceMajor);
    System.out.println("gender? " + student.gender);
  }
}
```

The following code has a compile error, because the local variables **`x`** and **`y`** are not initialized:

```
class TestLocalVariables {
  public static void main(String[] args) {
    int x; // x has no default value
    String y; // y has no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```

NullPointerException

Caution

`NullPointerException` is a common runtime error. It occurs when you invoke a method on a reference variable with a **`null`** value. Make sure you assign an object reference to the variable before invoking the method through the reference variable (see CheckPoint Question 9.5.5c).

9.5.3 Differences between Variables of Primitive Types and Reference Types

Every variable represents a memory location that holds a value. When you declare a variable, you are telling the compiler what type of value the variable can hold. For a variable of a primitive type, the value is of the primitive type. For a variable of a reference type, the value is a reference to where an object is located. For example, as shown in Figure 9.7, the value of **`int`** variable **`i`** is **`int`** value **`1`**, and the value of **`Circle`** object **`c`** holds a reference to where the contents of the **`Circle`** object are stored in memory.

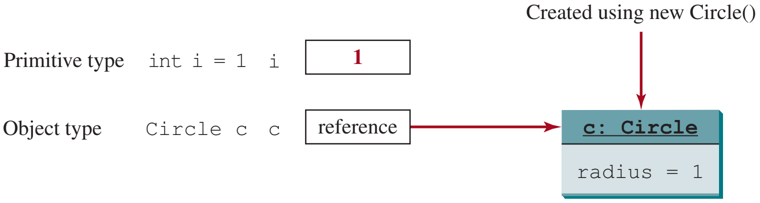

Created using new Circle()
Primitive type
int i = 1
i
1
Object type
Circle c
c
reference
c: Circle
radius = 1

FIGURE 9.7 A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an object is stored in memory.

When you assign one variable to another, the other variable is set to the same value. For a variable of a primitive type, the real value of one variable is assigned to the other variable. For a variable of a reference type, the reference of one variable is assigned to the other variable. As shown in Figure 9.8, the assignment statement `i = j` copies the contents of `j` into `i` for primitive variables. As shown in Figure 9.9, the assignment statement `c1 = c2` copies the reference of `c2` into `c1` for reference variables. After the assignment, variables `c1` and `c2` refer to the same object.

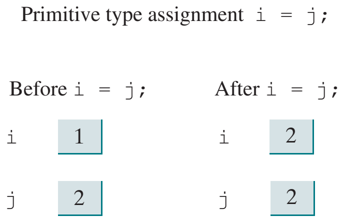

Primitive type assignment i = j;
Before i = j;
After i = j;
i
1
j
2
i
2
j
2

FIGURE 9.8 Primitive variable `j` is copied to variable `i`.

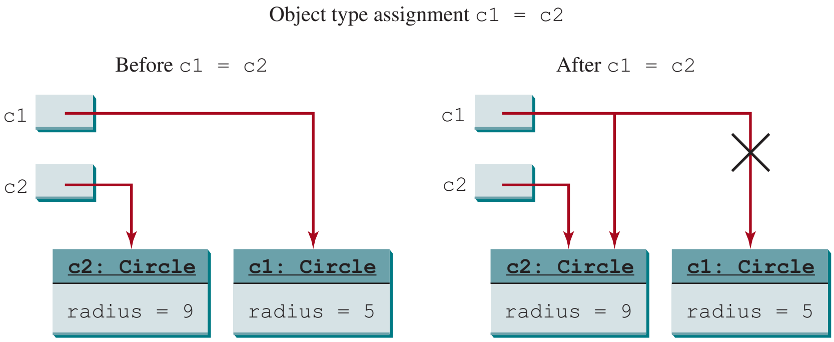

Object type assignment c1 = c2
Before c1 = c2
After c1 = c2
c1
c2
c2: Circle
radius = 9
c1: Circle
radius = 5
c1
c2
c2: Circle
radius = 9
c1: Circle
radius = 5

FIGURE 9.9 Reference variable `c2` is copied to variable `c1`.

Note

As illustrated in Figure 9.9, after the assignment statement `c1 = c2`, `c1` points to the same object referenced by `c2`. The object previously referenced by `c1` is no longer useful and therefore is now known as *garbage*. Garbage occupies memory space, so the Java runtime system detects garbage and automatically reclaims the space it occupies. This process is called *garbage collection*.

garbage

garbage collection

Tip

If you know that an object is no longer needed, you can explicitly assign `null` to a reference variable for the object. The JVM will automatically collect the space if the object is not referenced by any reference variable.

Check Point

9.5.1 Is an array an object or a primitive-type value? Can an array contain elements of an object type? Describe the default value for the elements of an array.

9.5.2 Which operator is used to access a data field or invoke a method from an object?

9.5.3 What is an anonymous object?

9.5.4 What is `NullPointerException`?

9.5.5 What is wrong with each of the following programs?

```
public class ShowErrors {
  public static void main(String[] args) {
    ShowErrors t = new ShowErrors();
    t.x();
  }
}
```

(a)

```
public class ShowErrors {
  public void method1() {
    Circle c;
    System.out.println("What is radius "
      + c.getRadius());
    c = new Circle();
  }
}
```

(b)

9.5.6 What is the output of the following code?

```
public class A {
  boolean x;

  public static void main(String[] args) {
    A a = new A();
    System.out.println(a.x);
  }
}
```

9.6 Using Classes from the Java Library

Key Point

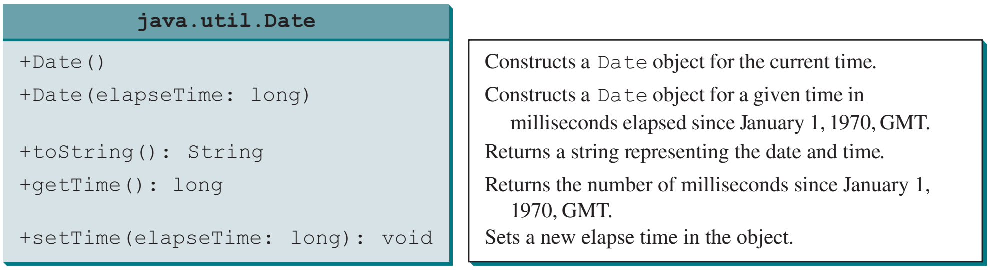
The Java API contains a rich set of classes for developing Java programs.

Listing 9.1 defined the `Circle` class and created objects from the class. You will frequently use the classes in the Java library to develop programs. This section gives some examples of the classes in the Java library.

9.6.1 The `Date` Class

java.util.Date class

In Listing 2.7, ShowCurrentTime.java, you learned how to obtain the current time using `System.currentTimeMillis()`. You used the division and remainder operators to extract the current second, minute, and hour. Java provides a system-independent encapsulation of date and time in the `java.util.Date` class, as shown in Figure 9.10.



java.util.Date	
+Date()	Constructs a Date object for the current time.
+Date(elapseTime: long)	Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
+toString(): String	Returns a string representing the date and time.
+getTime(): long	Returns the number of milliseconds since January 1, 1970, GMT.
+setTime(elapseTime: long): void	Sets a new elapse time in the object.

Figure 9.10 A `Date` object represents a specific date and time.

You can use the no-arg constructor in the **Date** class to create an instance for the current date and time, the **getTime()** method to return the elapsed time in milliseconds since January 1, 1970, GMT, and the **toString()** method to return the date and time as a string, For example, the following code:

```
java.util.Date date = new java.util.Date();
System.out.println("The elapsed time since Jan 1, 1970 is " +
  date.getTime() + " milliseconds");
System.out.println(date.toString());
```

create object

get elapsed time

invoke `toString`

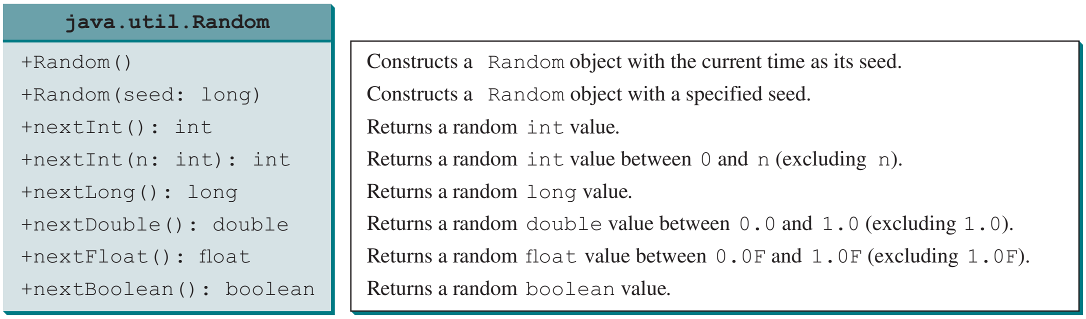
displays the output as follows:

```
The elapsed time since Jan 1, 1970 is 1324903419651 milliseconds
Mon Dec 26 07:43:39 EST 2011
```

The **Date** class has another constructor, **Date(long elapseTime)**, which can be used to construct a **Date** object for a given time in milliseconds elapsed since January 1, 1970, GMT.

9.6.2 The Random Class

You have used **Math.random()** to obtain a random **double** value between **0.0** and **1.0** (excluding **1.0**). Another way to generate random numbers is to use the **java.util.Random** class, as shown in Figure 9.11, which can generate a random **int**, **long**, **double**, **float**, and **boolean** value.



java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (excluding n).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (excluding 1.0).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (excluding 1.0F).
+nextBoolean(): boolean	Returns a random boolean value.

FIGURE 9.11 A **Random** object can be used to generate random values.

When you create a **Random** object, you have to specify a seed or use the default seed. A seed is a number used to initialize a random number generator. The no-arg constructor creates a **Random** object using the current elapsed time as its seed. If two **Random** objects have the same seed, they will generate identical sequences of numbers. For example, the following code creates two **Random** objects with the same seed, **3**:

```
Random generator1 = new Random(3);
System.out.print("From generator1: ");
for (int i = 0; i < 10; i++)
  System.out.print(generator1.nextInt(1000) + " ");

Random generator2 = new Random(3);
System.out.print("\nFrom generator2: ");
for (int i = 0; i < 10; i++)
  System.out.print(generator2.nextInt(1000) + " ");
```

The code generates the same sequence of random **int** values:

```
From generator1: 734 660 210 581 128 202 549 564 459 961
From generator2: 734 660 210 581 128 202 549 564 459 961
```

same sequence

Note

The ability to generate the same sequence of random values is useful in software testing and many other applications. In software testing, often you need to reproduce the test cases from a fixed sequence of random numbers.

`SecureRandom`

Note

You can generate random numbers using the **`java.security.SecureRandom`** class rather than the **`Random`** class. The random numbers generated from the **`Random`** are deterministic and they can be predicted by hackers. The random numbers generated from the **`SecureRandom`** class are nondeterministic and are secure.

9.6.3 The `Point2D` Class

Java API has a convenient **`Point2D`** class in the **`javafx.geometry`** package for representing a point in a two-dimensional plane. The UML diagram for the class is shown in Figure 9.12.

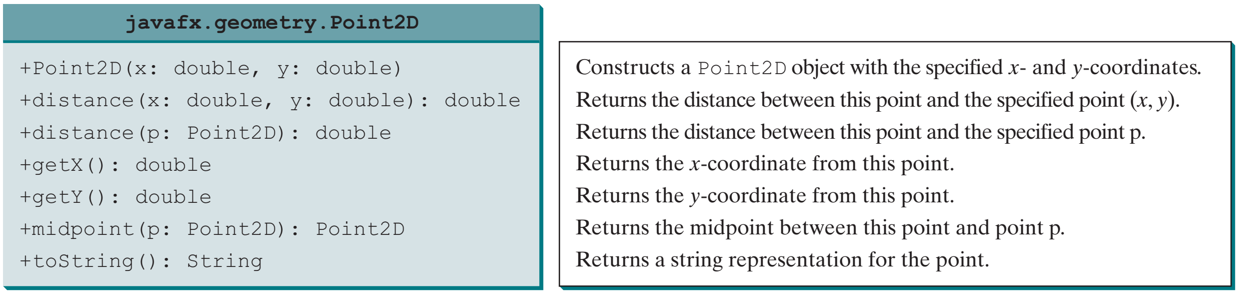

javafx.geometry.Point2D
+Point2D(x: double, y: double)
+distance(x: double, y: double): double
+distance(p: Point2D): double
+getX(): double
+getY(): double
+midpoint(p: Point2D): Point2D
+toString(): String
Constructs a Point2D object with the specified x- and y-coordinates.
Returns the distance between this point and the specified point (x, y).
Returns the distance between this point and the specified point p.
Returns the x-coordinate from this point.
Returns the y-coordinate from this point.
Returns the midpoint between this point and point p.
Returns a string representation for the point.

FIGURE 9.12 A `Point2D` object represents a point with *x*- and *y*-coordinates.

You can create a **`Point2D`** object for a point with the specified *x*- and *y*-coordinates, use the **`distance`** method to compute the distance from this point to another point, and use the **`toString()`** method to return a string representation of the point. Listing 9.5 gives an example of using this class.

LISTING 9.5 `TestPoint2D.java`

```
import java.util.Scanner;
import javafx.geometry.Point2D;

public class TestPoint2D {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    System.out.print("Enter point1's x-, y-coordinates: ");
    double x1 = input.nextDouble();
    double y1 = input.nextDouble();
    System.out.print("Enter point2's x-, y-coordinates: ");
    double x2 = input.nextDouble();
    double y2 = input.nextDouble();

    Point2D p1 = new Point2D(x1, y1);
    Point2D p2 = new Point2D(x2, y2);
    System.out.println("p1 is " + p1.toString());
    System.out.println("p2 is " + p2.toString());
    System.out.println("The distance between p1 and p2 is " +
      p1.distance(p2));
    System.out.println("The midpoint between p1 and p2 is " +
      p1.midpoint(p2).toString());
  }
}
```

create an object

invoke `toString()`

get distance

get midpoint

```
Enter point1's x-, y-coordinates: 1.5 5.5 ↵Enter
Enter point2's x-, y-coordinates: -5.3 -4.4 ↵Enter
p1 is Point2D [x = 1.5, y = 5.5]
p2 is Point2D [x = -5.3, y = -4.4]
The distance between p1 and p2 is 12.010412149464313
The midpoint between p1 and p2 is
Point2D [x = -1.9, y = 0.5499999999999998]
```

This program creates two objects of the `Point2D` class (lines 15 and 16). The `toString()` method returns a string that describes the object (lines 17 and 18). Invoking `p1.distance(p2)` returns the distance between the two points (line 20). Invoking `p1.midpoint(p2)` returns the midpoint between the two points (line 22).

Note

The `Point2D` class is defined in the `javafx.geometry` package, which is in the JavaFX module. To run this program, you need to install JavaFX. See Supplement II.F for installing and using JavaFX.

Check Point

9.6.1 How do you create a `Date` for the current time? How do you display the current time?

9.6.2 How do you create a `Point2D`? Suppose `p1` and `p2` are two instances of `Point2D`, how do you obtain the distance between the two points? How do you obtain the midpoint between the two points?

9.6.3 Which packages contain the classes `Date`, `Random`, `Point2D`, `System`, and `Math`?

9.7 Static Variables, Constants, and Methods

Key Point

A static variable is shared by all objects of the class. A static method cannot access instance members (i.e., instance data fields and methods) of the class.

Static vs. instance

instance variable

The data field `radius` in the `Circle` class is known as an *instance variable*. An instance variable is tied to a specific instance of the class; it is not shared among objects of the same class. For example, suppose that you create the following objects:

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
```

VideoNote

Static vs. instance

The `radius` in `circle1` is independent of the `radius` in `circle2` and is stored in a different memory location. Changes made to `circle1`'s `radius` do not affect `circle2`'s `radius`, and vice versa.

static variable

static method

If you want all the instances of a class to share data, use *static variables*, also known as *class variables*. Static variables store values for the variables in a common memory location. Because of this common location, if one object changes the value of a static variable, all objects of the same class are affected. Java supports static methods as well as static variables. *Static methods* can be called without creating an instance of the class.

Let's modify the `Circle` class by adding a static variable `numberOfObjects` to count the number of circle objects created. When the first object of this class is created, `numberOfObjects` is `1`. When the second object is created, `numberOfObjects` becomes `2`. The UML of the new circle class is shown in Figure 9.13. The `Circle` class defines the instance variable `radius` and the static variable `numberOfObjects`, the instance methods `getRadius`, `setRadius`, and `getArea`, and the static method `getNumberOfObjects`. (Note static variables and methods are underlined in the UML class diagram.)

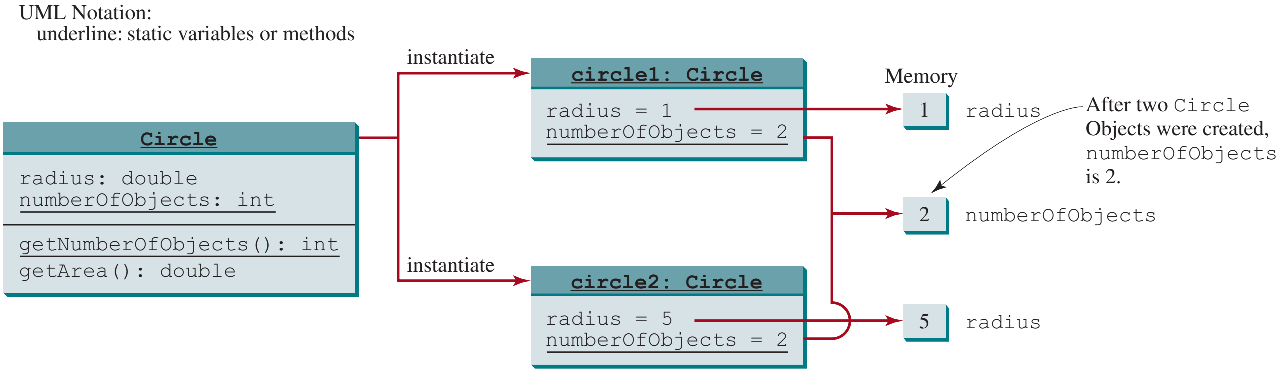

UML Notation:
underline: static variables or methods
instantiate
circle1: Circle
radius = 1
numberOfObjects = 2
Memory
1
radius
After two Circle Objects were created, numberOfObjects is 2.
Circle
radius: double
numberOfObjects: int
getNumberOfObjects(): int
getArea(): double
2
numberOfObjects
instantiate
circle2: Circle
radius = 5
numberOfObjects = 2
5
radius

FIGURE 9.13 Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class.

To declare a static variable or define a static method, put the modifier `static` in the variable or method declaration. The static variable `numberOfObjects` and the static method `getNumberOfObjects()` can be declared as follows:

declare static variable

```
static int numberOfObjects;
```

define static method

```
static int getNumberObjects() {
  return numberOfObjects;
}
```

declare constant

Constants in a class are shared by all objects of the class. Thus, constants should be declared as `final static`. For example, the constant `PI` in the `Math` class is defined as follows:

```
final static double PI = 3.14159265358979323846;
```

The new circle class is defined in Listing 9.6.

LISTING 9.6 Circle.java (for CircleWithStaticMembers)

```
public class Circle {
  /** The radius of the circle */
  double radius;

  /** The number of objects created */
  static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  Circle() {
    radius = 1;
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  Circle(double newRadius) {
    radius = newRadius;
    numberOfObjects++;
  }

  /** Return numberOfObjects */
  static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
```

static variable (line 6)

increase by 1 (line 11)

increase by 1 (line 17)

static method (line 21)

```
  double getArea() {
    return radius * radius * Math.PI;
  }
}
```

Method **`getNumberOfObjects()`** in **`Circle`** is a static method. All the methods in the **`Math`** class are static. The **`main`** method is static, too.

Instance methods (e.g., **`getArea()`**) and instance data (e.g., **`radius`**) belong to instances and can be used only after the instances are created. They are accessed via a reference variable. Static methods (e.g., **`getNumberOfObjects()`**) and static data (e.g., **`numberOfObjects`**) can be accessed from a reference variable or from their class name.

The program in Listing 9.7 demonstrates how to use instance and static variables and methods and illustrates the effects of using them.

LISTING 9.7 TestCircleWithStaticMembers.java

```
public class TestCircleWithStaticMembers {
  /** Main method */
  public static void main(String[] args) {
    System.out.println("Before creating objects");
    System.out.println("The number of Circle objects is " +
      Circle.numberOfObjects);

    // Create c1
    Circle c1 = new Circle(); // Use the Circle class in Listing 9.6

    // Display c1 BEFORE c2 is created
    System.out.println("\nAfter creating c1");
    System.out.println("c1: radius (" + c1.radius +
      ") and number of Circle objects (" +
      c1.numberOfObjects + ")");

    // Create c2
    Circle c2 = new Circle(5);

    // Modify c1
    c1.radius = 9;

    // Display c1 and c2 AFTER c2 was created
    System.out.println("\nAfter creating c2 and modifying c1");
    System.out.println("c1: radius (" + c1.radius +
      ") and number of Circle objects (" +
      c1.numberOfObjects + ")");
    System.out.println("c2: radius (" + c2.radius +
      ") and number of Circle objects (" +
      c2.numberOfObjects + ")");
  }
}
```

static variable

instance variable

static variable

instance variable

static variable

static variable

```
Before creating objects
The number of Circle objects is 0
After creating c1
c1: radius (1.0) and number of Circle objects (1)
After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

When you compile `TestCircleWithStaticMembers.java`, the Java compiler automatically compiles `Circle.java` if it has not been compiled since the last change.

Static variables and methods can be accessed without creating objects. Line 6 displays the number of objects, which is `0`, since no objects have been created.

The `main` method creates two circles `c1` and `c2` (lines 9 and18). The instance variable `radius` in `c1` is modified to become `9` (line 21). This change does not affect the instance variable `radius` in `c2`, since these two instance variables are independent. The static variable `numberOfObjects` becomes `1` after `c1` is created (line 9), and it becomes `2` after `c2` is created (line 18).

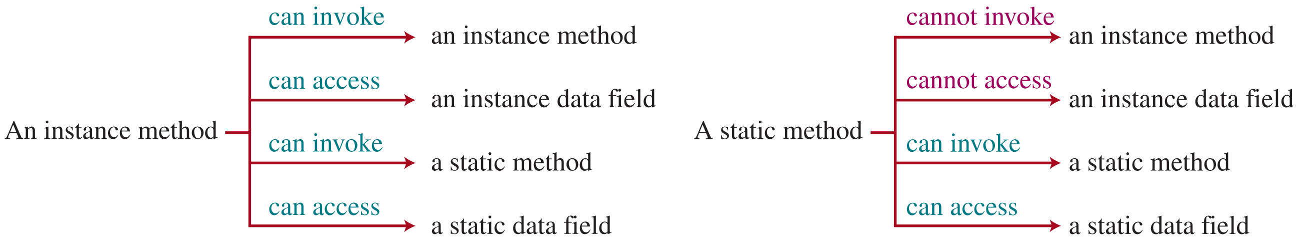
Note `PI` is a constant defined in `Math` and `Math.PI` references the constant. `c1.numberOfObjects` (line 27) and `c2.numberOfObjects` (line 30) are better replaced by `Circle.numberOfObjects`. This improves readability because other programmers can easily recognize the static variable. You can also replace `Circle.numberOfObjects` with `Circle.getNumberOfObjects()`.

use class name

Tip

Use `ClassName.methodName(arguments)` to invoke a static method and `ClassName.staticVariable` to access a static variable. This improves readability because this makes static methods and data easy to spot.

An instance method can invoke an instance or static method, and access an instance or static data field. A static method can invoke a static method and access a static data field. However, a static method cannot invoke an instance method or access an instance data field, since static methods and static data fields don't belong to a particular object. The relationship between static and instance members is summarized in the following diagram:


An instance method
can invoke
an instance method
can access
an instance data field
can invoke
a static method
can access
a static data field
A static method
cannot invoke
an instance method
cannot access
an instance data field
can invoke
a static method
can access
a static data field

For example, the following code is wrong.

```java
public class A {
  int i = 5;
  static int  k = 2;

  public static void main(String[] args) {
    int j = i; // Wrong because i is an instance variable
    m1(); // Wrong because m1() is an instance method
  }

  public void m1() {
    // Correct since instance and static variables and methods
    // can be used in an instance method
    i = i + k + m2(i, k);
  }

  public static int m2(int i, int j) {
    return (int)(Math.pow(i, j));
  }
}
```

Note if you replace the preceding code with the following new code, the program would be fine, because the instance data field `i` and method `m1` are now accessed from an object `a` (lines 7 and 8):

```
1   public class A {
2     int i = 5;
3     static int k = 2;
4
5     public static void main(String[] args) {
6       A a = new A();
7       int j = a.i; // OK, a.i accesses the object's instance variable
8       a.m1(); // OK, a.m1() invokes the object's instance method
9     }
10
11    public void m1() {
12      i = i + k + m2(i, k);
13    }
14
15    public static int m2(int i, int j) {
16      return (int)(Math.pow(i, j));
17    }
18  }
```

Design Guide

instance or static?

How do you decide whether a variable or a method should be instance or static? A variable or a method that is dependent on a specific instance of the class should be an instance variable or method. A variable or a method that is not dependent on a specific instance of the class should be a static variable or method. For example, every circle has its own radius, so the radius is dependent on a specific circle. Therefore, `radius` is an instance variable of the `Circle` class. Since the `getArea` method is dependent on a specific circle, it is an instance method. None of the methods in the `Math` class, such as `random`, `pow`, `sin`, and `cos`, is dependent on a specific instance. Therefore, these methods are static methods. The `main` method is static and can be invoked directly from a class.

Caution

common design error

It is a common design error to define an instance method that should have been defined as static. For example, the method `factorial(int n)` should be defined as static, as shown next, because it is independent of any specific instance.

```
public class Test {
  public int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

(a) Wrong design

```
public class Test {
  public static int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

(b) Correct design

Check Point

9.7.1 Suppose the class `F` is defined in (a). Let `f` be an instance of `F`. Which of the statements in (b) are correct?

```
public class F {
  int i;
  static String s;

  void imethod() {
  }

  static void smethod() {
  }
}
```

(a)

```
System.out.println(f.i);
System.out.println(f.s);
f.imethod();
f.smethod();
System.out.println(F.i);
System.out.println(F.s);
F.imethod();
F.smethod();
```

(b)

9.7.2 Add the `static` keyword in the place of `?` if appropriate.

```
public class Test {
  int count;

  public ? void main(String[] args) {
    ...
  }

  public ? int getCount() {
    return count;
  }

  public ? int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

9.7.3 Can you invoke an instance method or reference an instance variable from a static method? Can you invoke a static method or reference a static variable from an instance method? What is wrong in the following code?

```
public class C {
  Circle c = new Circle();

  public static void main(String[] args) {
    method1();
  }

  public void method1() {
    method2();
  }

  public static void method2() {
    System.out.println("What is radius " + c.getRadius());
  }
}
```

9.8 Visibility Modifiers

Key Point

Visibility modifiers can be used to specify the visibility of a class and its members.

package-private (or package-access)

You can use the `public` visibility modifier for classes, methods, and data fields to denote they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as *package-private* or *package-access*.

Note

using packages

Packages can be used to organize classes. To do so, you need to add the following line as the first noncomment and nonblank statement in the program:

```
package packageName;
```

If a class is defined without the package statement, it is said to be placed in the *default package*.

Java recommends that you place classes into packages rather than using a default package. For simplicity, however, this book uses default packages. For more information on packages, see Supplement III.E, Packages.

In addition to the `public` and default visibility modifiers, Java provides the `private` and `protected` modifiers for class members. This section introduces the `private` modifier. The `protected` modifier will be introduced in Section 11.14, The `protected` Data and Methods.

The `private` modifier makes methods and data fields accessible only from within its own class. Figure 9.14 illustrates how a public, default, and private data field or method in class `C1` can be accessed from a class `C2` in the same package, and from a class `C3` in a different package.

```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

public class C2 {
  void aMethod() {
    C1 c1 = new C1();
    can access c1.x;
    can access c1.y;
    cannot access c1.z;

    can invoke c1.m1();
    can invoke c1.m2();
    cannot invoke c1.m3();
  }
}
```

```
package p2;

public class C3 {
  void aMethod() {
    C1 c1 = new C1();
    can access c1.x;
    cannot access c1.y;
    cannot access c1.z;

    can invoke c1.m1();
    cannot invoke c1.m2();
    cannot invoke c1.m3();
  }
}
```

Figure 9.14 The private modifier restricts access to its defining class, the default modifier restricts access to a package, and the public modifier enables unrestricted access.

If a class is not defined as public, it can be accessed only within the same package. As shown in Figure 9.15, `C1` can be accessed from `C2`, but not from `C3`.

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

public class C3 {
  cannot access p1.C1;
  can access p1.C2;
}
```

Figure 9.15 A nonpublic class has package access.

inside access

The private modifier restricts private members from being accessed outside the class. However, there is no restriction on accessing members from inside the class. Therefore, objects instantiated in its own class can access its private members. As shown in Figure 9.16a, an object `c` of class `C` can access its private members, because `c` is defined inside its own class. However, in Figure 9.16b, an object `c` of class `C` cannot access its private members, because `c` is in the `Test` class.

```
public class C {
  private boolean x;

  public static void main(string[] args) {
    C c = new C();
    system.out.println(c.x);
    system.out.println(c.convert());
  }

  private int convert() {
    return x ? 1 : −1;
  }
}
```

(a) This is okay because object c is used inside the class C.

```
public class Test {
  public static void main(string[] args) {
    C c = new C();
    system.out.println(c.x);
    system.out.println(c.convert());
  }
}
```

(b) This is wrong because x and convert are private in class C.

Figure 9.16 An object can access its private members if it is defined in its own class.

Caution

The `private` modifier applies only to the members of a class. The `public` modifier can apply to a class or members of a class. Using the modifiers `public` and `private` on local variables would cause a compile error.

private constructor

Note

In most cases, the constructor should be public. However, if you want to prohibit the user from creating an instance of a class, use a *private constructor*. For example, there is no reason to create an instance from the `Math` class, because all of its data fields and methods are static. To prevent the user from creating objects from the `Math` class, the constructor in `java.lang.Math` is defined as follows:

```
private Math() {
}
```

9.9 Data Field Encapsulation

Making data fields private protects data and makes the class easy to maintain.

Key Point

Data field encapsulation

VideoNote

Data field encapsulation

The data fields `radius` and `numberOfObjects` in the `Circle` class in Listing 9.6 can be modified directly (e.g., `c1.radius = 5` or `Circle.numberOfObjects = 10`). This is not a good practice—for two reasons:

1. Data may be tampered with. For example, `numberOfObjects` is to count the number of objects created, but it may be mistakenly set to an arbitrary value (e.g., `Circle.numberOfObjects = 10`).
2. The class becomes difficult to maintain and vulnerable to bugs. Suppose that you want to modify the `Circle` class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the `Circle` class but also the programs that use it because the clients may have modified the radius directly (e.g., `c1.radius = –5`).

data field encapsulation

To prevent direct modifications of data fields, you should declare the data fields private, using the `private` modifier. This is known as *data field encapsulation*.

A private data field cannot be accessed by an object from outside the class that defines the private field. However, a client often needs to retrieve and modify a data field. To make a private data field accessible, provide a *getter* method to return its value. To enable a private data field to be updated, provide a *setter* method to set a new value. A getter method is also referred to as an *accessor* and a setter to a *mutator*. A getter method has the following signature:

getter (or accessor)

setter (or mutator)

```
public returnType getPropertyName()
```

If the `returnType` is `boolean`, the getter method should be defined as follows by convention:

boolean accessor

```
public boolean isPropertyName()
```

A setter method has the following signature:

```
public void setPropertyName(dataType propertyValue)
```

Let's create a new circle class with a private data-field radius and its associated accessor and mutator methods. The class diagram is shown in Figure 9.17. The new circle class is defined in Listing 9.8:

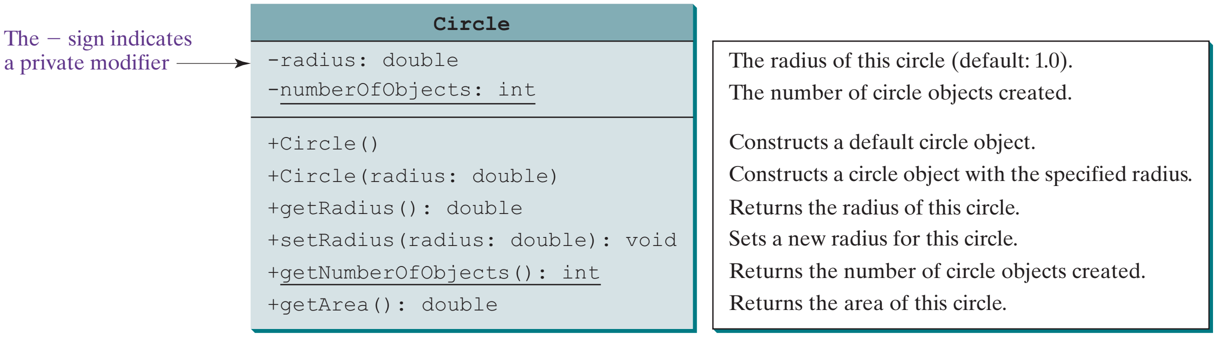

The − sign indicates
a private modifier
Circle
-radius: double
-numberOfObjects: int
+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getNumberOfObjects(): int
+getArea(): double
The radius of this circle (default: 1.0).
The number of circle objects created.
Constructs a default circle object.
Constructs a circle object with the specified radius.
Returns the radius of this circle.
Sets a new radius for this circle.
Returns the number of circle objects created.
Returns the area of this circle.

FIGURE 9.17 The `Circle` class encapsulates circle properties and provides getter/setter and other methods.

LISTING 9.8 `Circle.java(for CircleWithPrivateDataFields)`

```
public class Circle {
  /** The radius of the circle */
  private double radius = 1;

  /** The number of objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public Circle() {
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  public Circle(double newRadius) {
    radius = newRadius;
    numberOfObjects++;
```

encapsulate radius

encapsulate `numberOfObjects`

accessor method

mutator method

accessor method

```
17    }
18
19    /** Return radius */
20    public double getRadius() {
21      return radius;
22    }
23
24    /** Set a new radius */
25    public void setRadius(double newRadius) {
26      radius = (newRadius >= 0) ? newRadius : 0;
27    }
28
29    /** Return numberOfObjects */
30    public static int getNumberOfObjects() {
31      return numberOfObjects;
32    }
33
34    /** Return the area of this circle */
35    public double getArea() {
36      return radius * radius * Math.PI;
37    }
38  }
```

The **`getRadius()`** method (lines 20–22) returns the radius and the **`setRadius(newRadius)`** method (lines 25–27) sets a new radius for the object. If the new radius is negative, **`0`** is set as the radius for the object. Since these methods are the only ways to read and modify the radius, you have total control over how the **`radius`** property is accessed. If you have to change the implementation of these methods, you don't need to change the client programs. This makes the class easy to maintain.

Listing 9.9 gives a client program that uses the **`Circle`** class to create a **`Circle`** object, and modifies the radius using the **`setRadius`** method.

LISTING 9.9 TestCircleWithPrivateDataFields.java

invoke public method

invoke public method

invoke public method

```
public class TestCircleWithPrivateDataFields {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 5.0
    Circle myCircle = new Circle(5.0);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    // Increase myCircle's radius by 10%
    myCircle.setRadius(myCircle.getRadius() * 1.1);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    System.out.println("The number of objects created is "
      + Circle.getNumberOfObjects());
  }
}
```

The data field **`radius`** is declared private. Private data can be accessed only within their defining class, so you cannot use **`myCircle.radius`** in the client program. A compile error would occur if you attempted to access private data from a client.

Since **`numberOfObjects`** is private, it cannot be modified. This prevents tampering. For example, the user cannot set **`numberOfObjects`** to **`100`**. The only way to make it **`100`** is to create **`100`** objects of the **`Circle`** class.

Suppose you combined `TestCircleWithPrivateDataFields` and `Circle` into one class by moving the `main` method in `TestCircleWithPrivateDataFields` into `Circle`. Could you use `myCircle.radius` in the `main` method? See CheckPoint Question 9.9.3 for the answer.

Design Guide

To prevent data from being tampered with and to make the class easy to maintain, declare data fields private.

Note

From now on, all data fields should be declared private, and all constructors and methods should be defined public, unless specified otherwise.

Check Point

9.9.1 What is an accessor method? What is a mutator method? What are the naming conventions for accessor methods and mutator methods?

9.9.2 What are the benefits of data field encapsulation?

9.9.3 In the following code, `radius` is private in the `Circle` class, and `myCircle` is an object of the `Circle` class. Does the highlighted code cause any problems? If so, explain why.

```
public class Circle {
  private double radius = 1;

  /** Find the area of this circle */
  public double getArea() {
    return radius * radius * Math.PI;
  }

  public static void main(String[] args) {
    Circle myCircle = new Circle();
    System.out.println("Radius is " + myCircle.radius);
  }
}
```

9.10 Passing Objects to Methods

Passing an object to a method is to pass the reference of the object.

Key Point

You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object. The following code passes the `myCircle` object as an argument to the `printCircle` method:

pass an object

```
public class Test {
  public static void main(String[] args) {
    // Circle is defined in Listing 9.8
    Circle myCircle = new Circle(5.0);
    printCircle(myCircle);
  }

  public static void printCircle(Circle c) {
    System.out.println("The area of the circle of radius "
      + c.getRadius() + " is " + c.getArea());
  }
}
```

pass-by-value

Java uses exactly one mode of passing arguments: pass-by-value. In the preceding code, the value of `myCircle` is passed to the `printCircle` method. This value is a reference to a `Circle` object.

The program in Listing 9.10 demonstrates the difference between passing a primitive-type value and passing a reference value.

LISTING 9.10 TestPassObject.java

pass object

object parameter

```
 1 public class TestPassObject {
 2   /** Main method */
 3   public static void main(String[] args) {
 4     // Create a Circle object with radius 1
 5     Circle myCircle =
 6       new Circle(1); // Use the Circle class in Listing 9.8
 7
 8     // Print areas for radius 1, 2, 3, 4, and 5.
 9     int n = 5;
10     printAreas(myCircle, n);
11
12     // See myCircle.radius and times
13     System.out.println("\n" + "Radius is " + myCircle.getRadius());
14     System.out.println("n is " + n);
15   }
16
17   /** Print a table of areas for radius */
18   public static void printAreas(Circle c, int times) {
19     System.out.println("Radius \t\tArea");
20     while (times >= 1) {
21       System.out.println(c.getRadius() + "\t\t" + c.getArea());
22       c.setRadius(c.getRadius() + 1);
23       times--;
24     }
25   }
26 }
```

```
Radius        Area
1.0           3.141592653589793
2.0           12.566370614359172
3.0           28.274333882308138
4.0           50.26548245743669
5.0           78.53981633974483
Radius is 6.0
n is 5
```

The **Circle** class is defined in Listing 9.8. The program passes a **Circle** object **myCircle** and an integer value from **n** to invoke **printAreas(myCircle, n)** (line 10), which prints a table of areas for radii **1**, **2**, **3**, **4**, and **5**, as presented in the sample output.

Figure 9.18 shows the call stack for executing the methods in the program. Note the objects are stored in a heap (see Section 7.6).

When passing an argument of a primitive data type, the value of the argument is passed. In this case, the value of **n** (**5**) is passed to **times**. Inside the **printAreas** method, the content of **times** is changed; this does not affect the content of **n**.

pass-by-sharing

When passing an argument of a reference type, the reference of the object is passed. In this case, **c** contains a reference for the object that is also referenced via **myCircle**. Therefore, changing the properties of the object through **c** inside the **printAreas** method has the same effect as doing so outside the method through the variable **myCircle**. Pass-by-value on references can be best described semantically as *pass-by-sharing*; that is, the object referenced in the method is the same as the object being passed.

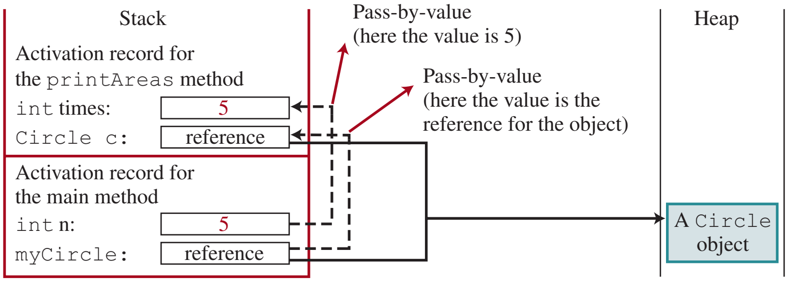

Stack
Activation record for the printAreas method
int times:
5
Circle c:
reference
Activation record for the main method
int n:
5
myCircle:
reference
Pass-by-value (here the value is 5)
Pass-by-value (here the value is the reference for the object)
Heap
A Circle object

FIGURE 9.18 The value of `n` is passed to `times`, and the reference to `myCircle` is passed to `c` in the `printAreas` method.

9.10.1 Describe the difference between passing a parameter of a primitive type and passing a parameter of a reference type. Show the output of the following programs:

Check Point

```
public class Test {
  public static void main(String[] args) {
    Count myCount = new  Count();
    int times = 0;

    for (int i = 0; i < 100; i++)
      increment(myCount, times);

    System.out.println("count is " + myCount.count);
    System.out.println("times is " + times);
  }

  public static void increment(Count c, int times) {
    c.count++;
    times++;
  }
}
```

```
public class Count {
  public int count;

  public Count (int c) {
    count = c;
  }

  public Count () {
    count = 1;
  }
}
```

9.10.2 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    Circle circle1 = new Circle(1);
    Circle circle2 = new Circle(2);

    swap1(circle1, circle2);
    System.out.println("After swap1: circle1 = " +
      circle1.radius + " circle2 = " + circle2.radius);

    swap2(circle1, circle2);
    System.out.println("After swap2: circle1 = " +
      circle1.radius + " circle2 = " + circle2.radius);
  }

  public static void swap1(Circle x, Circle y) {
    Circle temp = x;
    x = y;
    y = temp;
  }
```

```java
  public static void swap2(Circle x, Circle y) {
    double temp = x.radius;
    x.radius = y.radius;
    y.radius = temp;
  }
}

class Circle {
  double radius;

  Circle(double newRadius) {
    radius = newRadius;
  }
}
```

9.10.3 Show the output of the following code:

```java
public class Test {
  public static void main(String[] args) {
    int[] a = {1, 2};
    swap(a[0], a[1]);
    System.out.println("a[0] = " + a[0]
      + " a[1] = " + a[1]);
  }

  public static void swap(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
  }
}
```

(a)

```java
public class Test {
  public static void main(String[] args) {
    int[] a = {1, 2};
    swap(a);
    System.out.println("a[0] = " + a[0]
      + " a[1] = " + a[1]);
  }

  public static void swap(int[] a) {
    int temp = a[0];
    a[0] = a[1];
    a[1] = temp;
  }
}
```

(b)

```java
public class Test {
  public static void main(String[] args) {
    T t = new T();
    swap(t);
    System.out.println("e1 = " + t.e1
      + " e2 = " + t.e2);
  }

  public static void swap(T t) {
    int temp = t.e1;
    t.e1 = t.e2;
    t.e2 = temp;
  }
}

class T {
  int e1 = 1;
  int e2 = 2;
}
```

(c)

```java
public class Test {
  public static void main(String[] args) {
    T t1 = new T();
    T t2 = new T();
    System.out.println("t1's i = " +
      t1.i + " and j = " + t1.j);
    System.out.println("t2's i = " +
      t2.i + " and j = " + t2.j);
  }
}

class T {
  static int i = 0;
  int j = 0;

  T() {
    i++;
    j = 1;
  }
}
```

(d)

9.10.4 What is the output of the following programs?

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = null;
    m1(date);
    System.out.println(date);
  }

  public static void m1(Date date) {
    date = new Date();
  }
}
```

(a)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = new Date(7654321);
  }
}
```

(b)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date.setTime(7654321);
  }
}
```

(c)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = null;
  }
}
```

(d)

9.11 Array of Objects

An array can hold objects as well as primitive-type values.

Key Point

Chapter 7, Single-Dimensional Arrays, described how to create arrays of primitive-type elements. You can also create arrays of objects. For example, the following statement declares and creates an array of 10 `Circle` objects:

```
Circle[] circleArray = new Circle[10];
```

To initialize `circleArray`, you can use a `for` loop as follows:

```
for (int i = 0; i < circleArray.length; i++) {
  circleArray[i] = new Circle();
}
```

An array of objects is actually an *array of reference variables*. Thus, invoking `circleArray[1].getArea()` involves two levels of referencing, as shown in Figure 9.19. `circleArray` references the entire array, and `circleArray[1]` references a `Circle` object.

Note

When an array of objects is created using the `new` operator, each element in the array is a reference variable with a default value of `null`.

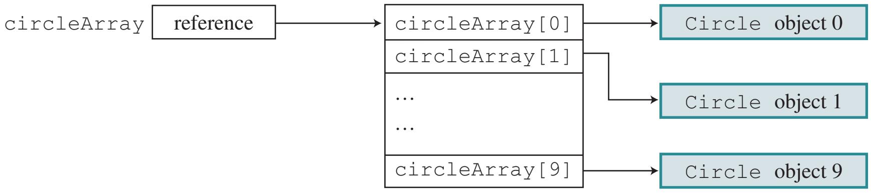


FIGURE 9.19 In an array of objects, an element of the array contains a reference to an object.

Listing 9.11 gives an example that demonstrates how to use an array of objects. The program summarizes the areas of an array of circles. The program creates **`circleArray`**, an array composed of five **`Circle`** objects; it then initializes circle radii with random values and displays the total area of the circles in the array.

LISTING 9.11 `TotalArea.java`

array of objects

return array of objects

pass array of objects

```java
public class TotalArea {
  /** Main method */
  public static void main(String[] args) {
    // Declare circleArray
    Circle[] circleArray;

    // Create circleArray
    circleArray = createCircleArray();

    // Print circleArray and total areas of the circles
    printCircleArray(circleArray);
  }

  /** Create an array of Circle objects */
  public static Circle[] createCircleArray() {
    Circle[] circleArray = new Circle[5];

    for (int i = 0; i < circleArray.length; i++) {
      circleArray[i] = new Circle(Math.random() * 100);
    }

    // Return Circle array
    return circleArray;
  }

  /** Print an array of circles and their total area */
  public static void printCircleArray(Circle[] circleArray) {
    System.out.printf("%-30s%-15s\n", "Radius", "Area");
    for (int i = 0; i < circleArray.length; i++) {
      System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
        circleArray[i].getArea());
    }

    System.out.println("____________________________________");

    // Compute and display the result
    System.out.printf("%-30s%-15f\n", "The total area of circles is",
      sum(circleArray));
  }

41    /** Add circle areas */
42    public static double sum(Circle[] circleArray) {
43      // Initialize sum
44      double sum = 0;
45
46      // Add areas to sum
47      for (int i = 0; i < circleArray.length; i++)
48        sum += circleArray[i].getArea();
49
50      return sum;
51    }
52  }
```

pass array of objects

```
Radius                         Area
70.577708                      15649.941866
44.152266                       6124.291736
24.867853                       1942.792644
 5.680718                        101.380949
36.734246                       4239.280350
_____________________________________________
The total area of circles is 28056.687544
```

The program invokes `createCircleArray()` (line 8) to create an array of five circle objects. Several circle classes were introduced in this chapter. This example uses the `Circle` class introduced in Section 9.9, Data Field Encapsulation.

The circle radii are randomly generated using the `Math.random()` method (line 19). The `createCircleArray` method returns an array of `Circle` objects (line 23). The array is passed to the `printCircleArray` method, which displays the radius and area of each circle and the total area of the circles.

The sum of the circle areas is computed by invoking the `sum` method (line 38), which takes the array of `Circle` objects as the argument and returns a `double` value for the total area.

9.11.1 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    java.util.Date[] dates = new java.util.Date[10];
    System.out.println(dates[0]);
    System.out.println(dates[0].toString());
  }
}
```

9.12 Immutable Objects and Classes

You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.

Key Point

Normally, you create an object and allow its contents to be changed later. However, occasionally it is desirable to create an object whose contents cannot be changed once the object has been created. We call such an object as *immutable object* and its class as *immutable class*. The `String` class, for example, is immutable. If you deleted the setter method in the `Circle` class in Listing 9.8, the class would be immutable because radius is private and cannot be changed without a setter method.

VideoNote
Immutable objects and this keyword

immutable object
immutable class

If a class is immutable, then all its data fields must be private and it cannot contain public setter methods for any data fields. A class with all private data fields and no mutators is not

`Student` class

necessarily immutable. For example, the following **`Student`** class has all private data fields and no setter methods, but it is not an immutable class:

```
public class Student {
  private int id;
  private String name;
  private java.util.Date dateCreated;

  public Student(int ssn, String newName) {
    id = ssn;
    name = newName;
    dateCreated = new java.util.Date();
  }

  public int getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public java.util.Date getDateCreated() {
    return dateCreated;
  }
}
```

As shown in the following code, the data field **`dateCreated`** is returned using the **`getDateCreated()`** method. This is a reference to a **`Date`** object. Through this reference, the content for **`dateCreated`** can be changed.

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student(111223333, "John");
    java.util.Date dateCreated = student.getDateCreated();
    dateCreated.setTime(200000); // Now dateCreated field is changed!
  }
}
```

For a class to be immutable, it must meet the following requirements:

- All data fields must be private.
- There can't be any mutator methods for data fields.
- No accessor methods can return a reference to a data field that is mutable.

Interested readers may refer to Supplement III.U for an extended discussion on immutable objects.

Check Point

9.12.1 If a class contains only private data fields and no setter methods, is the class immutable?

9.12.2 If all the data fields in a class are private and of primitive types, and the class doesn't contain any setter methods, is the class immutable?

9.12.3 Is the following class immutable?

```
public class A {
  private int[] values;

  public int[] getValues() {
        return values;
      }
    }
```

9.13 The Scope of Variables

The scope of instance and static variables is the entire class, regardless of where the variables are declared.

Key Point

Section 6.9 discussed local variables and their scope rules. Local variables are declared and used inside a method locally. This section discusses the scope rules of all the variables in the context of a class.

class's variables

Instance and static variables in a class are referred to as the *class's variables* or *data fields*. A variable defined inside a method is referred to as a *local variable*. The scope of a class's variables is the entire class, regardless of where the variables are declared. A class's variables and methods can appear in any order in the class, as shown in Figure 9.20a. The exception is when a data field is initialized based on a reference to another data field. In such cases, the other data field must be declared first, as shown in Figure 9.20b. For consistency, this book declares data fields at the beginning of the class.

```
public class Circle {
  public double getArea() {
    return radius * radius * Math.PI;
  }

  private double radius = 1;
}
```

(a)

(a) The variable **`radius`** and method **`getArea()`** can be declared in any order.

```
public class F {
  private int i;
  private int j = i + 1;
}
```

(b)

(b) **`i`** has to be declared before **`j`** because **`j`**'s initial value is dependent on **`i`**.

FIGURE 9.20 Members of a class can be declared in any order, with one exception.

You can declare a class's variable only once, but you can declare the same variable name in a method many times in different nonnesting blocks.

hidden variables

If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is *hidden*. For example, in the following program, **`x`** is defined both as an instance variable and as a local variable in the method:

```
public class F {
  private int x = 0; // Instance variable
  private int y = 0;

  public F() {
  }

  public void p() {
    int x = 1; // Local variable
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

What is the output for **`f.p()`**, where **`f`** is an instance of **`F`**? The output for **`f.p()`** is **`1`** for **`x`** and **`0`** for **`y`**. Here is why:

- **`x`** is declared as a data field with the initial value of **`0`** in the class, but it is also declared in the method **`p()`** with an initial value of **`1`**. The latter **`x`** is referenced in the **`System.out.println`** statement.
- **`y`** is declared outside the method **`p()`**, but **`y`** is accessible inside the method.

Tip
To avoid confusion and mistakes, do not use the names of instance or static variables as local variable names, except for method parameters. We will discuss hidden data fields by method parameters in the next section.

Check Point

9.13.1 What is the output of the following program?

```
public class Test {
  private static int i = 0;
  private static int j = 0;

  public static void main(String[] args) {
    int i = 2;
    int k = 3;

    {
      int j = 3;
      System.out.println("i + j is " + i + j);
    }

    k = i + j;
    System.out.println("k is " + k);
    System.out.println("j is " + j);
  }
}
```

9.14 The `this` Reference

Key Point

The keyword `this` *refers to the calling object. It can also be used inside a constructor to invoke another constructor of the same class.*

`this` keyword

VideoNote
The this keyword

When an instance method is called on an object, the `this` *keyword* is set to this object. You can use the `this` *keyword* to reference the object's instance members in the class. For example, the following code in (a) uses `this` to reference the object's `radius` and invokes its `getArea()` method explicitly. Some instructors prefer using the `this` keyword explicitly in the code, because it clearly distinguishes the instance variables from local variables. However, the `this` reference is normally omitted for brevity as shown in (b). Nevertheless, the `this` keyword is needed to reference a data field hidden by a method or constructor parameter, or to invoke an overloaded constructor.

```
public class Circle {
  private double radius;

  ...

  public double getArea() {
    return this.radius * this.radius
      * Math.PI;
  }

  public String toString() {
    return "radius: " + this.radius
      + "area: " + this.getArea();
  }
}
```

(a)

Equivalent

```
public class Circle {
  private double radius;

  ...

  public double getArea() {
    return radius * radius * Math.PI;
      * Math.PI;
  }

  public String toString() {
    return "radius: " + radius
      + "area: " + getArea();
  }
}
```

(b)

9.14.1 Using `this` to Reference Data Fields

It is a good practice to use the data field as the parameter name in a setter method or a constructor to make the code easy to read and to avoid creating unnecessary names. In this case, you need to use the `this` keyword to reference the data field in the setter method. For example, the `setRadius` method can be implemented as shown in (a). It would be wrong if it is implemented as shown in (b).

Refers to data field `radius` in this object.

```
private double radius;

public void setRadius(double radius) {
  this.radius = radius;
}
```

(a) Refers to data field radius in this object

Here, `radius` is the parameter in the method.

```
private double radius = 1;

public void setRadius(double radius) {
  radius = radius;
}
```

(b) Refers to parameter radius in the method.

The data field `radius` is hidden by the parameter `radius` in the setter method. You need to reference the data field name in the method using the syntax `this.radius`. A hidden static variable can be accessed simply by using the `ClassName.staticVariable` reference. A hidden instance variable can be accessed by using the keyword `this`, as shown in Figure 9.21a.

reference the data field

```
public class F {
  private int i = 5;
  private static double k = 0;

  public void setI(int i) {
    this.i = i;
  }

  public static
      void setK(double k) {
    F.k = k;

  }
  // other methods omitted
}
```

(a)

Suppose that f1 and f2 are two objects of F created as follows:.

```
F f1 = new F();
F f2 = new F();
```

Invoking `f1.setI(10)` is to execute `this.i = 10`, where this is an alias for `f1`

Invoking `f2.setI(45)` is to execute `this.i = 45`, where this is an alias for `f2`

Invoking `f2.setK(33)` is to execute `F.k = 33`. `setK` is a static method

(b)

FIGURE 9.21 The keyword `this` refers to the calling object that invokes the method.

The `this` keyword gives us a way to reference the object that invokes an instance method. To invoke `f1.setI(10)`, `this.i = i` is executed, which assigns the value of parameter `i` to the data field `i` of this calling object `f1`. The keyword `this` is an alias for `f1`, as shown in Figure 9.21b. The line `F.k = k` means the value in parameter `k` is assigned to the static data field `k` of the class, which is shared by all the objects of the class.

9.14.2 Using `this` to Invoke a Constructor

The `this` keyword can be used to invoke another constructor of the same class. For example, you can rewrite the `Circle` class as follows:

```
public class Circle {
  private double radius;
  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  ...
}
```

The `this` keyword is used to reference the data field radius of the object being constructed.

The `this` keyword is used to invoke another constructor.

The line `this(1.0)` in the second constructor invokes the first constructor with a `double` value argument.

Note

Java requires that the `this(arg-list)` statement appear first in the constructor before any other executable statements.

Tip

If a class has multiple constructors, it is better to implement them using `this(arg-list)` as much as possible. In general, a constructor with no or fewer arguments can invoke a constructor with more arguments using `this(arg-list)`. This syntax often simplifies coding and makes the class easier to read and to maintain.

Check Point

9.14.1 Describe the role of the `this` keyword.

9.14.2 What is wrong in the following code?

```
public class C {
  private int p;

  public C() {
    System.out.println("C's no-arg constructor invoked");
    this(0);
  }

  public C(int p) {
    p = p;
  }

  public void setP(int p) {
    p = p;
  }
}
```

9.14.3 What is wrong in the following code?

```
public class Test {
  private int id;

  public void m1() {
    this.id = 45;
  }
  public void m2() {
    Test.id = 45;
  }
}
```

KEY TERMS

accessor	347
action	324
anonymous object	333
attribute	324
behavior	324
class	324
class's variable	357
client	327
constructor	324
data field	324
data field encapsulation	346
default constructor	331
dot operator (`.`)	333
getter	347
immutable class	355
immutable object	355
instance	324
instance method	333
instance variable	333
instantiation	324
mutator	347
no-arg constructor	327
`null` value	334
object	324
object-oriented programming (OOP)	324
package-private (or package-access)	344
private constructor	346
property	324
public class	327
reference type	332
reference variable	332
setter	347
state	324
static method	339
static variable	339
`this` keyword	358
Unified Modeling Language (UML)	325

CHAPTER SUMMARY

1. A *class* is a template for *objects*. It defines the *properties* of objects and provides *constructors* for creating objects and methods for manipulating them.

2. A class is also a data type. You can use it to declare object *reference variables*. An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored.

3. An object is an *instance* of a class. You use the `new` operator to create an object and the *dot operator* (`.`) to access members of that object through its reference variable.

4. An *instance variable* or *method* belongs to an instance of a class. Its use is associated with individual instances. A *static variable* is a variable shared by all instances of the same class. A *static method* is a method that can be invoked without using instances.

5. Every instance of a class can access the class's static variables and methods. For clarity, however, it is better to invoke static variables and methods using `ClassName.variable` and `ClassName.method`.

6. Visibility modifiers specify how the class, method, and data are accessed. A `public` class, method, or data is accessible to all clients. A `private` method or data is accessible only inside the class.

7. You can provide a getter (accessor) method or a setter (mutator) method to enable clients to see or modify the data.

8. A getter method has the signature `public returnType getPropertyName()`. If the `returnType` is `boolean`, the getter method should be defined as `public boolean isPropertyName()`. A setter method has the signature `public void setPropertyName(dataType propertyValue)`.

9. All parameters are passed to methods using pass-by-value. For a parameter of a primitive type, the actual value is passed; for a parameter of a *reference type*, the reference for the object is passed.

10. A Java array is an object that can contain primitive-type values or object-type values. When an array of objects is created, its elements are assigned the default value of `null`.

11. Once it is created, an *immutable object* cannot be modified. To prevent users from modifying an object, you can define *immutable classes*.

12. The scope of instance and static variables is the entire class, regardless of where the variables are declared. Instance and static variables can be declared anywhere in the class. For consistency, they are declared at the beginning of the class in this book.

13. The keyword `this` can be used to refer to the calling object. It can also be used inside a constructor to invoke another constructor of the same class.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

three objectives

Pedagogical Note

The exercises in Chapters 9–13 help you to achieve three objectives:

1. Design classes and draw UML class diagrams.
2. Implement classes from the UML.
3. Use classes to develop applications.

Students can download solutions for the UML diagrams for the even-numbered exercises from the Companion Website and instructors can download all solutions from the Instructor Website.

Starting from Section 9.7, all data fields should be declared private and all constructors and methods should be defined public unless specified otherwise.

Sections 9.2–9.5

9.1 (*The* `Rectangle` *class*) Following the example of the `Circle` class in Section 9.2, design a class named `Rectangle` to represent a rectangle. The class contains:

- Two `double` data fields named `width` and `height` that specify the width and height of the rectangle. The default values are `1` for both `width` and `height`.
- A no-arg constructor that creates a default rectangle.
- A constructor that creates a rectangle with the specified `width` and `height`.
- A method named `getArea()` that returns the area of this rectangle.
- A method named `getPerimeter()` that returns the perimeter.

Draw the UML diagram for the class then implement the class. Write a test program that creates two `Rectangle` objects—one with width `4` and height `40`, and the other with width `3.5` and height `35.9`. Display the width, height, area, and perimeter of each rectangle in this order.

9.2 (*The* `Stock` *class*) Following the example of the `Circle` class in Section 9.2, design a class named `Stock` that contains:

- A string data field named `symbol` for the stock's symbol.
- A string data field named `name` for the stock's name.
- A `double` data field named `previousClosingPrice` that stores the stock price for the previous day.
- A `double` data field named `currentPrice` that stores the stock price for the current time.
- A constructor that creates a stock with the specified symbol and name.
- A method named `getChangePercent()` that returns the percentage changed from `previousClosingPrice` to `currentPrice`.

Draw the UML diagram for the class then implement the class. Write a test program that creates a `Stock` object with the stock symbol `ORCL`, the name `Oracle Corporation`, and the previous closing price of `34.5`. Set a new current price to `34.35` and display the price-change percentage.

Section 9.6

***9.3** (*Use the* `Date` *class*) Write a program that creates a `Date` object, sets its elapsed time to `10000`, `100000`, `1000000`, `10000000`, `100000000`, `1000000000`, `10000000000`, and `100000000000`, and displays the date and time using the `toString()` method, respectively.

***9.4** (*Use the* `Random` *class*) Write a program that creates a `Random` object with seed `1000` and displays the first 50 random integers between `0` and `100` using the `nextInt(100)` method.

***9.5** (*Use the* `GregorianCalendar` *class*) Java API has the `GregorianCalendar` class in the `java.util` package, which you can use to obtain the year, month, and day of a date. The no-arg constructor constructs an instance for the current date, and the methods `get(GregorianCalendar.YEAR)`, `get(GregorianCalendar.MONTH)`, and `get(GregorianCalendar.DAY_OF_MONTH)` return the year, month, and day. Write a program to perform two tasks:

1. Display the current year, month, and day.
2. The `GregorianCalendar` class has the `setTimeInMillis(long)`, which can be used to set a specified elapsed time since January 1, 1970. Set the value to `1234567898765L` and display the year, month, and day.

Sections 9.7–9.9

***9.6** (*Stopwatch*) Design a class named `StopWatch`. The class contains:

- Private data fields `startTime` and `endTime` with getter methods.
- A no-arg constructor that initializes `startTime` with the current time.
- A method named `start()` that resets the `startTime` to the current time.
- A method named `stop()` that sets the `endTime` to the current time.
- A method named `getElapsedTime()` that returns the elapsed time for the stopwatch in milliseconds.

Draw the UML diagram for the class then implement the class. Write a test program that measures the execution time of sorting 100,000 numbers using selection sort.

9.7 (*The* `Account` *class*) Design a class named `Account` that contains:

- A private `int` data field named `id` for the account (default `0`).
- A private `double` data field named `balance` for the account (default `0`).

- A private `double` data field named `annualInterestRate` that stores the current interest rate (default `0`). Assume that all accounts have the same interest rate.
- A private `Date` data field named `dateCreated` that stores the date when the account was created.
- A no-arg constructor that creates a default account.
- A constructor that creates an account with the specified id and initial balance.
- The accessor and mutator methods for `id`, `balance`, and `annualInterestRate`.
- The accessor method for `dateCreated`.
- A method named `getMonthlyInterestRate()` that returns the monthly interest rate.
- A method named `getMonthlyInterest()` that returns the monthly interest.
- A method named `withdraw` that withdraws a specified amount from the account.
- A method named `deposit` that deposits a specified amount to the account.

Draw the UML diagram for the class then implement the class. (*Hint*: The method `getMonthlyInterest()` is to return monthly interest, not the interest rate. Monthly interest is `balance * monthlyInterestRate`. `monthlyInterestRate` is `annualInterestRate / 12`. Note `annualInterestRate` is a percentage, for example 4.5%. You need to divide it by 100.)

Write a test program that creates an `Account` object with an account ID of 1122, a balance of $20,000, and an annual interest rate of 4.5%. Use the `withdraw` method to withdraw $2,500, use the `deposit` method to deposit $3,000, and print the balance, the monthly interest, and the date when this account was created.

VideoNote
The Fan class

9.8 (*The `Fan` class*) Design a class named `Fan` to represent a fan. The class contains:

- Three constants named `SLOW`, `MEDIUM`, and `FAST` with the values `1`, `2`, and `3` to denote the fan speed.
- A private `int` data field named `speed` that specifies the speed of the fan (the default is `SLOW`).
- A private `boolean` data field named `on` that specifies whether the fan is on (the default is `false`).
- A private `double` data field named `radius` that specifies the radius of the fan (the default is `5`).
- A string data field named `color` that specifies the color of the fan (the default is `blue`).
- The accessor and mutator methods for all four data fields.
- A no-arg constructor that creates a default fan.
- A method named `toString()` that returns a string description for the fan. If the fan is on, the method returns the fan speed, color, and radius in one combined string. If the fan is not on, the method returns the fan color and radius along with the string "fan is off" in one combined string.

Draw the UML diagram for the class then implement the class. Write a test program that creates two `Fan` objects. Assign maximum speed, radius `10`, color `yellow`, and turn it on to the first object. Assign medium speed, radius `5`, color `blue`, and turn it off to the second object. Display the objects by invoking their `toString` method.

****9.9** (*Geometry: n-sided regular polygon*) In an *n*-sided regular polygon, all sides have the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). Design a class named `RegularPolygon` that contains:

- A private `int` data field named `n` that defines the number of sides in the polygon with default value `3`.

- A private `double` data field named `side` that stores the length of the side with default value `1`.
- A private `double` data field named `x` that defines the *x*-coordinate of the polygon's center with default value `0`.
- A private `double` data field named `y` that defines the *y*-coordinate of the polygon's center with default value `0`.
- A no-arg constructor that creates a regular polygon with default values.
- A constructor that creates a regular polygon with the specified number of sides and length of side, centered at (`0`, `0`).
- A constructor that creates a regular polygon with the specified number of sides, length of side, and *x*- and *y*-coordinates.
- The accessor and mutator methods for all data fields.
- The method `getPerimeter()` that returns the perimeter of the polygon.
- The method `getArea()` that returns the area of the polygon. The formula for computing the area of a regular polygon is

$$\text{Area} = \frac{n \times s^2}{4 \times \tan\left(\dfrac{\pi}{n}\right)}.$$

Draw the UML diagram for the class then implement the class. Write a test program that creates three `RegularPolygon` objects, created using the no-arg constructor, using `RegularPolygon(6, 4)`, and using `RegularPolygon(10, 4, 5.6, 7.8)`. For each object, display its perimeter and area.

***9.10** (*Algebra: quadratic equations*) Design a class named `QuadraticEquation` for a quadratic equation $ax^2 + bx + c = 0$. The class contains:

- Private data fields `a`, `b`, and `c` that represent three coefficients.
- A constructor with the arguments for `a`, `b`, and `c`.
- Three getter methods for `a`, `b`, and `c`.
- A method named `getDiscriminant()` that returns the discriminant, which is $b^2 - 4ac$.
- The methods named `getRoot1()` and `getRoot2()` for returning two roots of the equation

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

These methods are useful only if the discriminant is nonnegative. Let these methods return `0` if the discriminant is negative.

Draw the UML diagram for the class then implement the class. Write a test program that prompts the user to enter values for *a*, *b*, and *c* and displays the result based on the discriminant. If the discriminant is positive, display the two roots. If the discriminant is 0, display the one root. Otherwise, display "The equation has no roots." See Programming Exercise 3.1 for sample runs.

***9.11** (*Algebra: 2 × 2 linear equations*) Design a class named `LinearEquation` for a 2 × 2 system of linear equations:

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \qquad x = \frac{ed - bf}{ad - bc} \qquad y = \frac{af - ec}{ad - bc}$$

The class contains:

- Private data fields `a`, `b`, `c`, `d`, `e`, and `f`.
- A constructor with the arguments for `a`, `b`, `c`, `d`, `e`, and `f`.
- Six getter methods for `a`, `b`, `c`, `d`, `e`, and `f`.
- A method named `isSolvable()` that returns true if $ad - bc$ is not 0.
- Methods `getX()` and `getY()` that return the solution for the equation.

Draw the UML diagram for the class then implement the class. Write a test program that prompts the user to enter `a`, `b`, `c`, `d`, `e`, and `f` and displays the result. If $ad - bc$ is 0, report that “The equation has no solution.” See Programming Exercise 3.3 for sample runs.

**9.12 (*Geometry: intersecting point*) Suppose two line segments intersect. The two endpoints for the first line segment are (`x1`, `y1`) and (`x2`, `y2`) and for the second line segment are (`x3`, `y3`) and (`x4`, `y4`). Write a program that prompts the user to enter these four endpoints and displays the intersecting point. As discussed in Programming Exercise 3.25, the intersecting point can be found by solving a linear equation. Use the `LinearEquation` class in Programming Exercise 9.11 to solve this equation. See Programming Exercise 3.25 for sample runs.

**9.13 (*The `Location` class*) Design a class named `Location` for locating a maximal value and its location in a two-dimensional array. The class contains public data fields `row`, `column`, and `maxValue` that store the maximal value and its indices in a two-dimensional array with `row` and `column` as `int` types and `maxValue` as a `double` type.

Write the following method that returns the location of the largest element in a two-dimensional array:

```
public static Location locateLargest(double[][] a)
```

The return value is an instance of `Location`. Write a test program that prompts the user to enter a two-dimensional array and displays the location of the largest element in the array. Here is a sample run:

```
Enter the number of rows and columns in the array: 3 4 ↵Enter
Enter the array:
23.5 35 2 10 ↵Enter
4.5 3 45 3.5 ↵Enter
35 44 5.5 9.6 ↵Enter
The location of the largest element is 45 at (1, 2)
```

CHAPTER

10

OBJECT-ORIENTED THINKING

Objectives

- To apply class abstraction to develop software (§10.2).
- To explore the differences between the procedural paradigm and object-oriented paradigm (§10.3).
- To discover the relationships between classes (§10.4).
- To design programs using the object-oriented paradigm (§§10.5 and 10.6).
- To create objects for primitive values using the wrapper classes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, and `Boolean`) (§10.7).
- To simplify programming using automatic conversion between primitive types and wrapper class types (§10.8).
- To use the `BigInteger` and `BigDecimal` classes for computing very large numbers with arbitrary precisions (§10.9).
- To use the `String` class to process immutable strings (§10.10).
- To use the `StringBuilder` and `StringBuffer` classes to process mutable strings (§10.11).

10.1 Introduction

Key Point

The focus of this chapter is on class design and to explore the differences between procedural programming and object-oriented programming.

The preceding chapter introduced objects and classes. You learned how to define classes, create objects, and use objects. This book's approach is to teach problem solving and fundamental programming techniques before object-oriented programming. This chapter shows how procedural and object-oriented programming differ. You will see the benefits of object-oriented programming and learn to use it effectively.

Our focus here is on class design. We will use several examples to illustrate the advantages of the object-oriented approach. The examples involve designing new classes and using them in applications and introducing new classes in the Java API.

10.2 Class Abstraction and Encapsulation

Key Point

Class abstraction is separation of class implementation from the use of a class. The details of implementation are encapsulated and hidden from the user. This is known as class encapsulation.

class abstraction

class's contract

class encapsulation

abstract data type

In Chapter 6, you learned about method abstraction and used it in stepwise refinement. Java provides many levels of abstraction, and *class abstraction* separates class implementation from how the class is used. The creator of a class describes the functions of the class and lets the user know how the class can be used. The collection of public constructors, methods, and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the *class's contract*. As shown in Figure 10.1, the user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user. This is called *class encapsulation*. For example, you can create a `Circle` object and find the area of the circle without knowing how the area is computed. For this reason, a class is also known as an *abstract data type* (ADT).

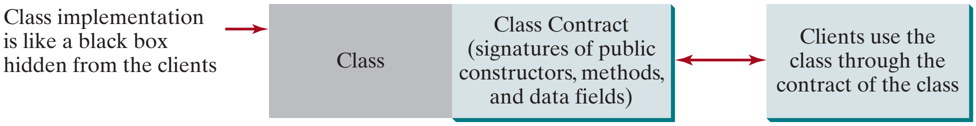


FIGURE 10.1 Class abstraction separates class implementation from the use of the class.

Class abstraction and encapsulation are two sides of the same coin. Many real-life examples illustrate the concept of class abstraction. Consider, for instance, building a computer system. Your personal computer has many components—a CPU, memory, disk, motherboard, fan, and so on. Each component can be viewed as an object that has properties and methods. To get the components to work together, you need to know only how each component is used and how it interacts with the others. You don't need to know how the components work internally. The internal implementation is encapsulated and hidden from you. You can build a computer without knowing how a component is implemented.

The computer-system analogy precisely mirrors the object-oriented approach. Each component can be viewed as an object of the class for the component. For example, you might have a class that models all kinds of fans for use in a computer, with properties such as fan size and speed and methods such as start and stop. A specific fan is an instance of this class with specific property values.

As another example, consider getting a loan. A specific loan can be viewed as an object of a `Loan` class. The interest rate, loan amount, and loan period are its data properties and computing the monthly and total payments are its methods. When you buy a car, a loan object is created by instantiating the class with your loan interest rate, loan amount, and loan period. You can then use the methods to find the monthly payment and total payment of your loan. As a user of the **Loan** class, you don't need to know how these methods are implemented.

VideoNote
the Loan class

Listing 2.9, ComputeLoan.java, presented a program for computing loan payments. That program cannot be reused in other programs because the code for computing the payments is in the `main` method. One way to fix this problem is to define static methods for computing the monthly payment and the total payment. However, this solution has limitations. Suppose that you wish to associate a date with the loan. There is no good way to tie a date with a loan without using objects. The traditional procedural programming paradigm is action-driven, and data are separated from actions. The object-oriented programming paradigm focuses on objects, and actions are defined along with the data in objects. To tie a date with a loan, you can define a loan class with a date along with the loan's other properties as data fields. A loan object now contains data and actions for manipulating and processing data, and the loan data and actions are integrated in one object. Figure 10.2 shows the UML class diagram for the **Loan** class.

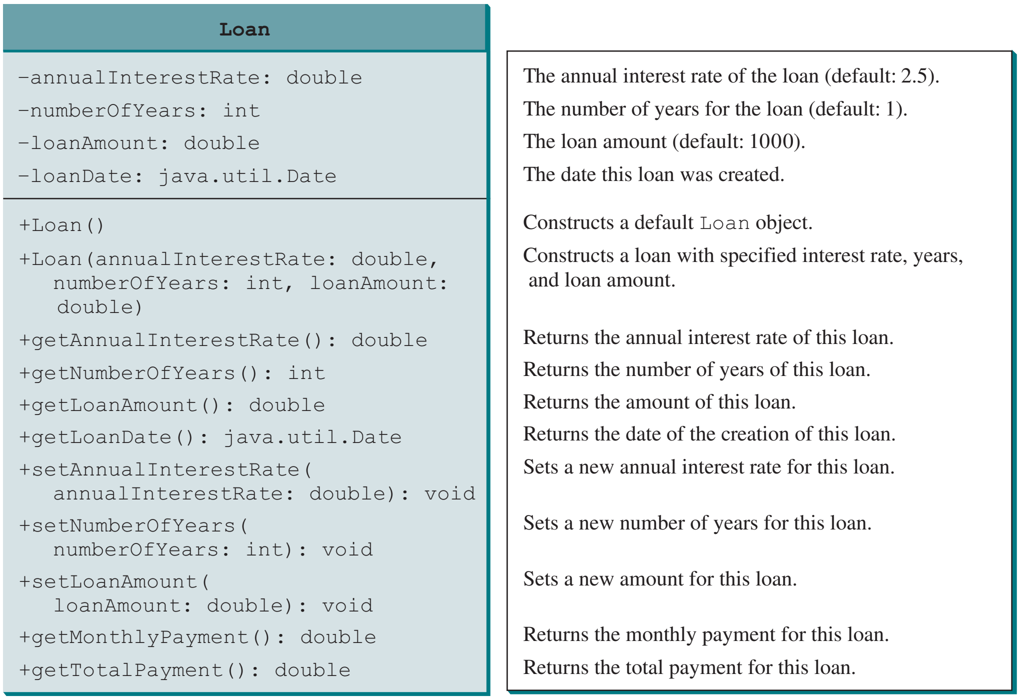

Loan
-annualInterestRate: double
-numberOfYears: int
-loanAmount: double
-loanDate: java.util.Date
+Loan()
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)
+getAnnualInterestRate(): double
+getNumberOfYears(): int
+getLoanAmount(): double
+getLoanDate(): java.util.Date
+setAnnualInterestRate(annualInterestRate: double): void
+setNumberOfYears(numberOfYears: int): void
+setLoanAmount(loanAmount: double): void
+getMonthlyPayment(): double
+getTotalPayment(): double
The annual interest rate of the loan (default: 2.5).
The number of years for the loan (default: 1).
The loan amount (default: 1000).
The date this loan was created.
Constructs a default Loan object.
Constructs a loan with specified interest rate, years, and loan amount.
Returns the annual interest rate of this loan.
Returns the number of years of this loan.
Returns the amount of this loan.
Returns the date of the creation of this loan.
Sets a new annual interest rate for this loan.
Sets a new number of years for this loan.
Sets a new amount for this loan.
Returns the monthly payment for this loan.
Returns the total payment for this loan.

FIGURE 10.2 The **Loan** class models the properties and behaviors of loans.

The UML diagram in Figure 10.2 serves as the contract for the **Loan** class. Throughout this book, you will play the roles of both class user and class developer. Remember that a class user can use the class without knowing how the class is implemented.

Assume the **Loan** class is available. The program in Listing 10.1 uses that class.

LISTING 10.1 `TestLoanClass.java`

```java
import java.util.Scanner;

public class TestLoanClass {
  /** Main method */
  public static void main(String[] args) {
```

create `loan` object

invoke instance method

invoke instance method

```
 6      // Create a Scanner
 7      Scanner input = new Scanner(System.in);
 8
 9      // Enter annual interest rate
10      System.out.print(
11        "Enter annual interest rate, for example, 8.25: ");
12      double annualInterestRate = input.nextDouble();
13
14      // Enter number of years
15      System.out.print("Enter number of years as an integer: ");
16      int numberOfYears = input.nextInt();
17
18      // Enter loan amount
19      System.out.print("Enter loan amount, for example, 120000.95: ");
20      double loanAmount = input.nextDouble();
21
22      // Create a Loan object
23      Loan loan =
24        new Loan(annualInterestRate, numberOfYears, loanAmount);
25
26      // Display loan date, monthly payment, and total payment
27      System.out.printf("The loan was created on %s\n" +
28        "The monthly payment is %.2f\nThe total payment is %.2f\n",
29        loan.getLoanDate().toString(), loan.getMonthlyPayment(),
30        loan.getTotalPayment());
31    }
32  }
```

```
Enter annual interest rate, for example, 8.25: 2.5 ↵Enter
Enter number of years as an integer: 5 ↵Enter
Enter loan amount, for example, 120000.95: 1000 ↵Enter
The loan was created on Sat Jun 16 21:12:50 EDT 2012
The monthly payment is 17.75
The total payment is 1064.84
```

The `main` method reads the interest rate, the payment period (in years), and the loan amount; creates a `Loan` object; then obtains the monthly payment (line 29) and the total payment (line 30) using the instance methods in the `Loan` class.

The `Loan` class can be implemented as in Listing 10.2.

LISTING 10.2 Loan.java

no-arg constructor

```
 1  public class Loan {
 2    private double annualInterestRate;
 3    private int numberOfYears;
 4    private double loanAmount;
 5    private java.util.Date loanDate;
 6
 7    /** Default constructor */
 8    public Loan() {
 9      this(2.5, 1, 1000);
10    }
11
```

```java
  /** Construct a loan with specified annual interest rate,
      number of years, and loan amount
    */
  public Loan(double annualInterestRate, int numberOfYears,
      double loanAmount) {
    this.annualInterestRate = annualInterestRate;
    this.numberOfYears = numberOfYears;
    this.loanAmount = loanAmount;
    loanDate = new java.util.Date();
  }

  /** Return annualInterestRate */
  public double getAnnualInterestRate() {
    return annualInterestRate;
  }

  /** Set a new annualInterestRate */
  public void setAnnualInterestRate(double annualInterestRate) {
    this.annualInterestRate = annualInterestRate;
  }

  /** Return numberOfYears */
  public int getNumberOfYears() {
    return numberOfYears;
  }

  /** Set a new numberOfYears */
  public void setNumberOfYears(int numberOfYears) {
    this.numberOfYears = numberOfYears;
  }

  /** Return loanAmount */
  public double getLoanAmount() {
    return loanAmount;
  }

  /** Set a new loanAmount */
  public void setLoanAmount(double loanAmount) {
    this.loanAmount = loanAmount;
  }

  /** Find monthly payment */
  public double getMonthlyPayment() {
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 –
      (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
    return monthlyPayment;
  }

  /** Find total payment */
  public double getTotalPayment() {
    double totalPayment = getMonthlyPayment() * numberOfYears * 12;
    return totalPayment;
  }

  /** Return loan date */
  public java.util.Date getLoanDate() {
    return loanDate;
  }
}
```

constructor

From a class developer's perspective, a class is designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through constructors, properties, and methods.

The `Loan` class contains two constructors, four getter methods, three setter methods, and the methods for finding the monthly payment and the total payment. You can construct a `Loan` object by using the no-arg constructor or the constructor with three parameters: annual interest rate, number of years, and loan amount. When a loan object is created, its date is stored in the `loanDate` field. The `getLoanDate` method returns the date. The methods—`getAnnualInterest`, `getNumberOfYears`, and `getLoanAmount`—return the annual interest rate, payment years, and loan amount, respectively. All the data properties and methods in this class are tied to a specific instance of the `Loan` class. Therefore, they are instance variables and methods.

Important Pedagogical Tip

Use the UML diagram for the `Loan` class shown in Figure 10.2 to write a test program that uses the `Loan` class even though you don't know how the `Loan` class is implemented. This has three benefits:

- It demonstrates that developing a class and using a class are two separate tasks.
- It enables you to skip the complex implementation of certain classes without interrupting the sequence of this book.
- It is easier to learn how to implement a class if you are familiar with it by using the class.

For all the class examples from now on, create an object from the class and try using its methods before turning your attention to its implementation.

Check Point

10.2.1 If you redefine the `Loan` class in Listing 10.2 without setter methods, is the class immutable?

10.3 Thinking in Objects

Key Point

The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects.

Chapters 1 through 8 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. Knowing these techniques lays a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software. This section improves the solution for a problem introduced in Chapter 3 using the object-oriented approach. From these improvements, you will gain insight into the differences between procedural and object-oriented programming, and see the benefits of developing reusable code using objects and classes.

Listing 3.4, ComputeAndInterpretBMI.java, presented a program for computing the body mass index (BMI). The code cannot be reused in other programs, because the code is in the `main` method. To make it reusable, define a static method to compute body mass index as follows:

```
public static double getBMI(double weight, double height)
```

This method is useful for computing body mass index for a specified weight and height. However, it has limitations. Suppose you need to associate the weight and height with a person's name and birth date. You could declare separate variables to store these values, but these values would not be tightly coupled. The ideal way to couple them is to create an object that contains them all. Since these values are tied to individual objects, they should be stored in instance data fields. You can define a class named `BMI` as shown in Figure 10.3.

VideoNote
The `BMI` class

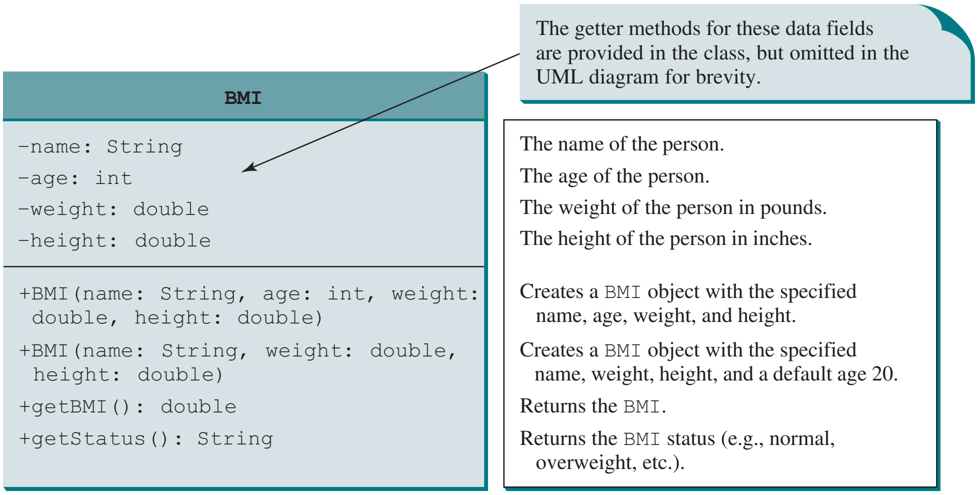

The getter methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
BMI
-name: String
-age: int
-weight: double
-height: double
+BMI(name: String, age: int, weight: double, height: double)
+BMI(name: String, weight: double, height: double)
+getBMI(): double
+getStatus(): String
The name of the person.
The age of the person.
The weight of the person in pounds.
The height of the person in inches.
Creates a BMI object with the specified name, age, weight, and height.
Creates a BMI object with the specified name, weight, height, and a default age 20.
Returns the BMI.
Returns the BMI status (e.g., normal, overweight, etc.).

Figure 10.3 The **BMI** class encapsulates BMI information.

Assume the **BMI** class is available. Listing 10.3 gives a test program that uses this class.

Listing 10.3 UseBMIClass.java

```
 1  public class UseBMIClass {
 2    public static void main(String[] args) {
 3      BMI bmi1 = new BMI("Kim Yang", 18, 145, 70);
 4      System.out.println("The BMI for " + bmi1.getName() + " is "
 5        + bmi1.getBMI() + " " + bmi1.getStatus());
 6
 7      BMI bmi2 = new BMI("Susan King", 215, 70);
 8      System.out.println("The BMI for " + bmi2.getName() + " is "
 9        + bmi2.getBMI() + " " + bmi2.getStatus());
10    }
11  }
```

create an object
invoke instance method

create an object
invoke instance method

```
The BMI for Kim Yang is 20.81 Normal
The BMI for Susan King is 30.85 Obese
```

Line 3 creates the object **bmi1** for **Kim Yang**, and line 7 creates the object **bmi2** for **Susan King**. You can use the instance methods **getName()**, **getBMI()**, and **getStatus()** to return the BMI information in a **BMI** object.

The **BMI** class can be implemented as in Listing 10.4.

Listing 10.4 BMI.java

```
public class BMI {
  private String name;
  private int age;
  private double weight; // in pounds
  private double height; // in inches
  public static final double KILOGRAMS_PER_POUND = 0.45359237;
  public static final double METERS_PER_INCH = 0.0254;

  public BMI(String name, int age, double weight, double height) {
    this.name = name;
```

constructor

constructor

getBMI

getStatus

```
    this.age = age;
    this.weight = weight;
    this.height = height;
  }

  public BMI(String name, double weight, double height) {
    this(name, 20, weight, height);
  }

  public double getBMI() {
    double bmi = weight * KILOGRAMS_PER_POUND /
      ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
    return Math.round(bmi * 100) / 100.0;
  }

  public String getStatus() {
    double bmi = getBMI();
    if (bmi < 18.5)
      return "Underweight";
    else if (bmi < 25)
      return "Normal";
    else if (bmi < 30)
      return "Overweight";
    else
      return "Obese";
  }

  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }

  public double getWeight() {
    return weight;
  }

  public double getHeight() {
    return height;
  }
}
```

The mathematical formula for computing the BMI using weight and height is given in Section 3.8. The instance method `getBMI()` returns the BMI. Since the weight and height are instance data fields in the object, the `getBMI()` method can use these properties to compute the BMI for the object.

The instance method `getStatus()` returns a string that interprets the BMI. The interpretation is also given in Section 3.8.

procedural vs. object-oriented paradigms

This example demonstrates the advantages of the object-oriented paradigm over the procedural paradigm. The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

In procedural programming, data and operations on the data are separate, and this methodology requires passing data to methods. Object-oriented programming places data and the operations that pertain to them in an object. This approach solves many of the problems inherent in procedural programming. The object-oriented programming approach organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Using objects improves software reusability and makes programs easier to develop and easier to maintain. Programming in Java involves thinking in terms of objects; a Java program can be viewed as a collection of cooperating objects.

10.3.1 Is the `BMI` class defined in Listing 10.4 immutable?

Check Point

10.4 Class Relationships

Key Point

To design classes, you need to explore the relationships among classes. The common relationships among classes are association, aggregation, composition, *and* inheritance.

This section explores association, aggregation, and composition. The inheritance relationship will be introduced in Chapter 11.

10.4.1 Association

association

Association is a general binary relationship that describes an activity between two classes. For example, a student taking a course is an association between the `Student` class and the `Course` class, and a faculty member teaching a course is an association between the `Faculty` class and the `Course` class. These associations can be represented in UML graphical notation, as shown in Figure 10.4.

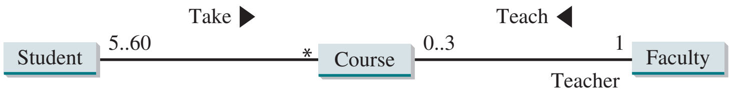


FIGURE 10.4 This UML diagram shows that a student may take any number of courses, a faculty member may teach at most three courses, a course may have from 5 to 60 students, and a course is taught by only one faculty member.

An association is illustrated by a solid line between two classes with an optional label that describes the relationship. In Figure 10.4, the labels are *Take* and *Teach*. Each relationship may have an optional small black triangle that indicates the direction of the relationship. In this figure, the ▶ indicates that a student takes a course (as opposed to a course taking a student).

Each class involved in the relationship may have a role name that describes the role it plays in the relationship. In Figure 10.4, *teacher* is the role name for `Faculty`.

multiplicity

Each class involved in an association may specify a *multiplicity*, which is placed at the side of the class to specify how many of the class's objects are involved in the relationship in UML. A multiplicity could be a number or an interval that specifies how many of the class's objects are involved in the relationship. The character `*` means an unlimited number of objects, and the interval `m..n` indicates that the number of objects is between `m` and `n`, inclusively. In Figure 10.4, each student may take any number of courses, and each course must have at least 5 and at most 60 students. Each course is taught by only one faculty member, and a faculty member may teach from 0 to 3 courses per semester.

In Java code, you can implement associations by using data fields and methods. For example, the relationships in Figure 10.4 may be implemented using the classes in Figure 10.5.

The relation "a student takes a course" is implemented using the **`addCourse`** method in the **`Student`** class and the **`addStudent`** method in the **`Course`** class. The relation "a faculty teaches a course" is implemented using the **`addCourse`** method in the **`Faculty`** class and the **`setFaculty`** method in the **`Course`** class. The **`Student`** class may use a list to store the courses that the student is taking, the **`Faculty`** class may use a list to store the courses that the faculty is teaching, and the **`Course`** class may use a list to store students enrolled in the course and a data field to store the instructor who teaches the course.

```
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

```
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

Figure 10.5 The association relations are implemented using data fields and methods in classes.

many possible implementations

Note

There are many possible ways to implement relationships. For example, the student and faculty information in the **`Course`** class can be omitted, since they are already in the **`Student`** and **`Faculty`** class. Likewise, if you don't need to know the courses a student takes or a faculty member teaches, the data field **`courseList`** and the **`addCourse`** method in **`Student`** or **`Faculty`** can be omitted.

10.4.2 Aggregation and Composition

aggregation
aggregating object
aggregating class
aggregated object
aggregated class
composition

Aggregation is a special form of association that represents an ownership relationship between two objects. Aggregation models *has-a* relationships. The owner object is called an *aggregating object*, and its class is called an *aggregating class*. The subject object is called an *aggregated object*, and its class is called an *aggregated class*.

We refer aggregation between two objects as *composition* if the existence of the aggregated object is dependent on the aggregating object. In other words, if a relationship is composition, the aggregated object cannot exist on its own. For example, "a student has a name" is a composition relationship between the **`Student`** class and the **`Name`** class because **`Name`** is dependent on **`Student`**, whereas "a student has an address" is an aggregation relationship between the **`Student`** class and the **`Address`** class because an address can exist by itself. Composition implies exclusive ownership. One object owns another object. When the owner object is destroyed, the dependent object is destroyed as well. In UML, a filled diamond is attached to an aggregating class (in this case, **`Student`**) to denote the composition relationship with an aggregated class (**`Name`**), and an empty diamond is attached to an aggregating class (**`Student`**) to denote the aggregation relationship with an aggregated class (**`Address`**), as shown in Figure 10.6.

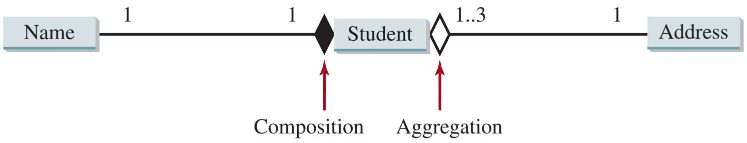


Figure 10.6 Each student has a name and an address.

In Figure 10.6, each student has only one multiplicity—address—and each address can be shared by up to **`3`** students. Each student has one name, and the name is unique for each student.

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationships in Figure 10.6 may be implemented using the classes in Figure 10.7. The relation "a student has a name" and "a student has an address" are implemented in the data field `name` and `address` in the `Student` class.

```
public class Name {
  ...
}
```

Aggregated class

```
public class Student {
  private Name name;
  private Address address;
  ...
}
```

Aggregating class

```
public class Address  {
  ...
}
```

Aggregated class

FIGURE 10.7 The composition relations are implemented using data fields in classes.

Aggregation may exist between objects of the same class. For example, a person may have a supervisor. This is illustrated in Figure 10.8.

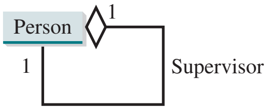
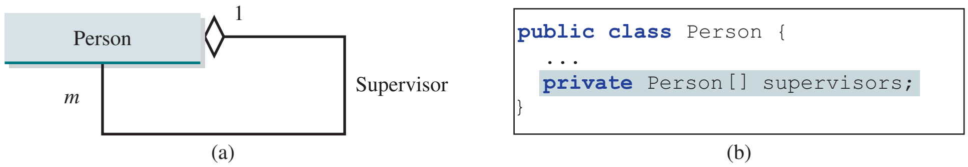


FIGURE 10.8 A person may have a supervisor.

In the relationship "a person has a supervisor," a supervisor can be represented as a data field in the `Person` class, as follows:

```
public class Person {
  // The type for the data is the class itself
  private Person supervisor;

  ...
}
```

If a person can have several supervisors, as shown in Figure 10.9a, you may use an array to store supervisors, as shown in Figure 10.9b.



(a)

```
public class Person {
  ...
  private Person[] supervisors;
}
```

(b)

FIGURE 10.9 A person can have several supervisors.

aggregation or composition

Important Note

Since aggregation and composition relationships are represented using classes in the same way, we will not differentiate them and call both compositions for simplicity.

Check Point

10.4.1 What are common relationships among classes?

10.4.2 What is association? What is aggregation? What is composition?

10.4.3 What is UML notation of aggregation and composition?

10.4.4 Why both aggregation and composition are together referred to as composition?

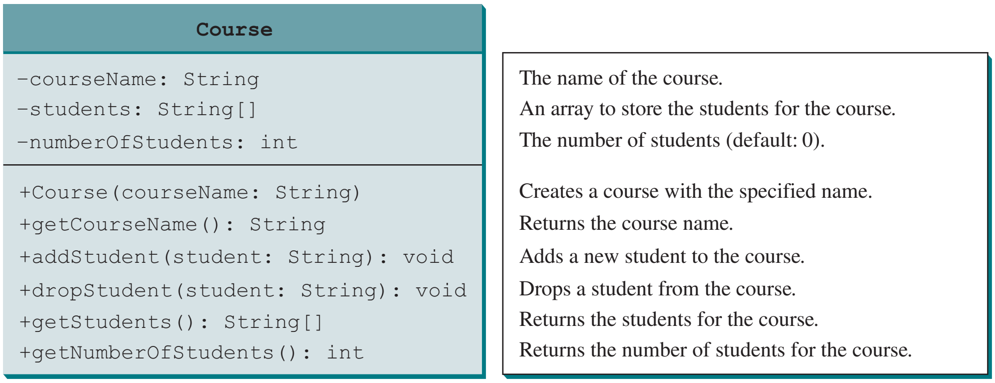
10.5 Case Study: Designing the **Course** Class

Key Point

This section designs a class for modeling courses.

This book's philosophy is *teaching by example and learning by doing*. The book provides a wide variety of examples to demonstrate object-oriented programming. This section and the next offer additional examples on designing classes.

Suppose you need to process course information. Each course has a name and has students enrolled. You should be able to add/drop a student to/from the course. You can use a class to model the courses, as shown in Figure 10.10.



Course	
-courseName: String	The name of the course.
-students: String[]	An array to store the students for the course.
-numberOfStudents: int	The number of students (default: 0).
+Course(courseName: String)	Creates a course with the specified name.
+getCourseName(): String	Returns the course name.
+addStudent(student: String): void	Adds a new student to the course.
+dropStudent(student: String): void	Drops a student from the course.
+getStudents(): String[]	Returns the students for the course.
+getNumberOfStudents(): int	Returns the number of students for the course.

Figure 10.10 The **Course** class models the courses.

A **Course** object can be created using the constructor **Course(String name)** by passing a course name. You can add students to the course using the **addStudent(String student)** method, drop a student from the course using the **dropStudent(String student)** method, and return all the students in the course using the **getStudents()** method. Suppose that the **Course** class is available; Listing 10.5 gives a test class that creates two courses and adds students to them.

Listing 10.5 TestCourse.java

create a Course

add a Student

number of students

return students

```java
public class TestCourse {
  public static void main(String[] args) {
    Course course1 = new Course("Data Structures");
    Course course2 = new Course("Database Systems");

    course1.addStudent("Peter Jones");
    course1.addStudent("Kim Smith");
    course1.addStudent("Anne Kennedy");

    course2.addStudent("Peter Jones");
    course2.addStudent("Steve Smith");

    System.out.println("Number of students in course1: "
      + course1.getNumberOfStudents());
    String[] students = course1.getStudents();
    for (int i = 0; i < course1.getNumberOfStudents(); i++)
      System.out.print(students[i] + ", ");

    System.out.println();
    System.out.print("Number of students in course2: "
      + course2.getNumberOfStudents());
  }
}
```

```
Number of students in course1: 3
Peter Jones, Kim Smith, Anne Kennedy,
Number of students in course2: 2
```

The `Course` class is implemented in Listing 10.6. It uses an array to store the students in the course. For simplicity, assume the maximum course enrollment is `100`. The array is created using `new String[100]` in line 3. The `addStudent` method (line 10) adds a student to the array. Whenever a new student is added to the course, `numberOfStudents` is increased (line 12). The `getStudents` method returns the array. The `dropStudent` method (line 27) is left as an exercise.

LISTING 10.6 Course.java

```
 1  public class Course {
 2    private String courseName;
 3    private String[] students = new String[100];
 4    private int numberOfStudents;
 5
 6    public Course(String courseName) {
 7      this.courseName = courseName;
 8    }
 9
10    public void addStudent(String student) {
11      students[numberOfStudents] = student;
12      numberOfStudents++;
13    }
14
15    public String[] getStudents() {
16      return students;
17    }
18
19    public int getNumberOfStudents() {
20      return numberOfStudents;
21    }
22
23    public String getCourseName() {
24      return courseName;
25    }
26
27    public void dropStudent(String student) {
28      // Left as an exercise in Programming Exercise 10.9
29    }
30  }
```

create students

add a course

return students

number of students

The array size is fixed to be `100` (line 3), so you cannot have more than 100 students in the course. You can improve the class by automatically increasing the array size in Programming Exercise 10.9.

When you create a `Course` object, an array object is created. A `Course` object contains a reference to the array. For simplicity, you can say the `Course` object contains the array.

The user can create a `Course` object and manipulate it through the public methods `addStudent`, `dropStudent`, `getNumberOfStudents`, and `getStudents`. However, the user doesn't need to know how these methods are implemented. The `Course` class encapsulates the internal implementation. This example uses an array to store students, but you could use a different data structure to store `students`. The program that uses `Course` does not need to change as long as the contract of the public methods remains unchanged.

Check Point

10.5.1 Replace the statement in line 17 in Listing 10.5, TestCourse.java, so the loop displays each student name followed by a comma except the last student name.

10.6 Case Study: Designing a Class for Stacks

Key Point

This section designs a class for modeling stacks.

stack

Recall that a *stack* is a data structure that holds data in a last-in, first-out fashion, as shown in Figure 10.11.

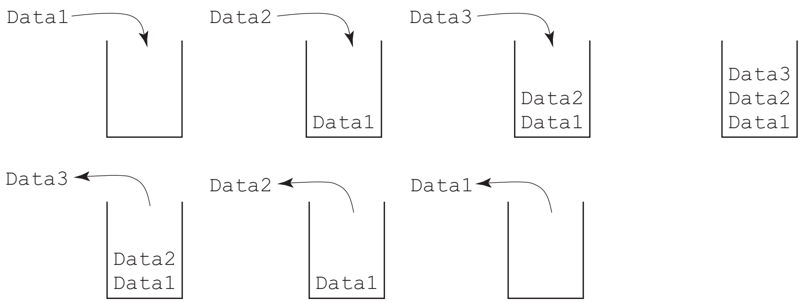
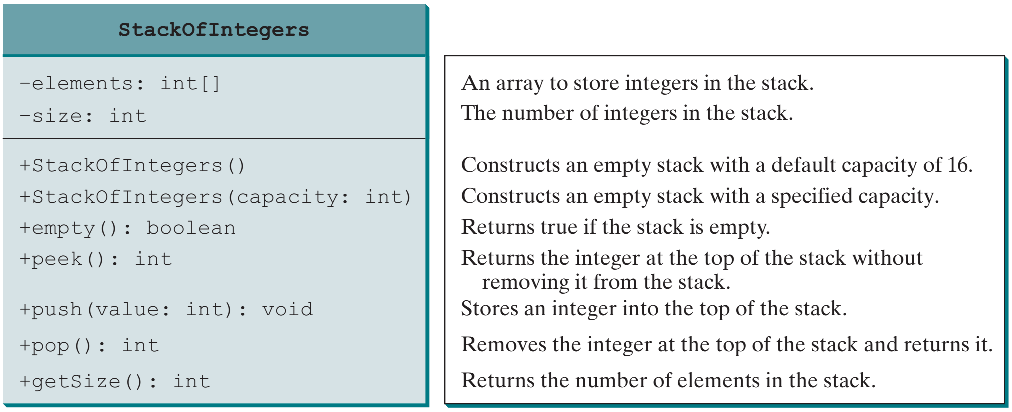


FIGURE 10.11 A stack holds data in a last-in, first-out fashion.

Stacks have many applications. For example, the compiler uses a stack to process method invocations. When a method is invoked, its parameters and local variables are pushed into a stack. When a method calls another method, the new method's parameters and local variables are pushed into the stack. When a method finishes its work and returns to its caller, its associated space is released from the stack.

VideoNote
The `StackOfIntegers` class

You can define a class to model stacks. For simplicity, assume the stack holds the `int` values. Thus, name the stack class `StackOfIntegers`. The UML diagram for the class is shown in Figure 10.12.



StackOfIntegers

`-elements: int[]`	An array to store integers in the stack.
`-size: int`	The number of integers in the stack.
`+StackOfIntegers()`	Constructs an empty stack with a default capacity of 16.
`+StackOfIntegers(capacity: int)`	Constructs an empty stack with a specified capacity.
`+empty(): boolean`	Returns true if the stack is empty.
`+peek(): int`	Returns the integer at the top of the stack without removing it from the stack.
`+push(value: int): void`	Stores an integer into the top of the stack.
`+pop(): int`	Removes the integer at the top of the stack and returns it.
`+getSize(): int`	Returns the number of elements in the stack.

FIGURE 10.12 The `StackOfIntegers` class encapsulates the stack storage and provides the operations for manipulating the stack.

Suppose the class is available. The test program in Listing 10.7 uses the class to create a stack (line 3), store 10 integers **0**, **1**, **2**, . . . , and **9** (line 6), and displays them in reverse order (line 9).

LISTING 10.7 TestStackOfIntegers.java

create a stack

```
1  public class TestStackOfIntegers {
2    public static void main(String[] args) {
3      StackOfIntegers stack = new StackOfIntegers();
```

```

    for (int i = 0; i < 10; i++)
      stack.push(i);

    while (!stack.empty())
      System.out.print(stack.pop() + " ");
  }
}
```

push to stack

pop from stack

```
9 8 7 6 5 4 3 2 1 0
```

How do you implement the **`StackOfIntegers`** class? The elements in the stack are stored in an array named **`elements`**. When you create a stack, the array is also created. The no-arg constructor creates an array with the default capacity of **`16`**. The variable **`size`** counts the number of elements in the stack, and **`size – 1`** is the index of the element at the top of the stack, as shown in Figure 10.13. For an empty stack, **`size`** is **`0`**.

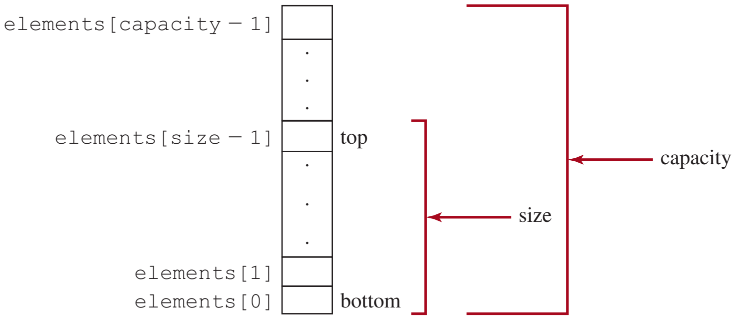


FIGURE 10.13 The **`StackOfIntegers`** uses an array to store the elements in a stack.

The **`StackOfIntegers`** class is implemented in Listing 10.8. The methods **`empty()`**, **`peek()`**, **`pop()`**, and **`getSize()`** are easy to implement. To implement **`push(int value)`**, assign **`value`** to **`elements[size]`** if **`size < capacity`** (line 24). If the stack is full (i.e., **`size >= capacity`**), create a new array of twice the current capacity (line 19), copy the contents of the current array to the new array (line 20), and assign the reference of the new array to the current array in the stack (line 21). Now you can add the new value to the array (line 24).

LISTING 10.8 `StackOfIntegers.java`

```
 1 public class StackOfIntegers {
 2   private int[] elements;
 3   private int size;
 4   public static final int DEFAULT_CAPACITY = 16;
 5
 6   /** Construct a stack with the default capacity 16 */
 7   public StackOfIntegers() {
 8     this(DEFAULT_CAPACITY);
 9   }
10
11   /** Construct a stack with the specified capacity */
12   public StackOfIntegers(int capacity) {
13     elements = new int[capacity];
14   }
15
```

max capacity 16

double the capacity

add to stack

```
  /** Push a new integer to the top of the stack */
  public void push(int value) {
    if (size >= elements.length) {
      int[] temp = new int[elements.length * 2];
      System.arraycopy(elements, 0, temp, 0, elements.length);
      elements = temp;
    }

    elements[size++] = value;
  }

  /** Return and remove the top element from the stack */
  public int pop() {
    return elements[--size];
  }

  /** Return the top element from the stack */
  public int peek() {
    return elements[size - 1];
  }

  /** Test whether the stack is empty */
  public boolean empty() {
    return size == 0;
  }

  /** Return the number of elements in the stack */
  public int getSize() {
    return size;
  }
}
```

Check Point

10.6.1 What happens when invoking the `pop()` method on a stack while size is `0`?

10.7 Processing Primitive Data Type Values as Objects

Key Point

A primitive-type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API.

why wrapper class?

Owing to performance considerations, primitive data type values are not objects in Java. Because of the overhead of processing objects, the language's performance would be adversely affected if primitive data type values were treated as objects. However, many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or wrap, a primitive data type value into an object (e.g., wrapping an `int` into an `Integer` object, wrapping a `double` into a `Double` object, and wrapping a `char` into a `Character` object). By using a wrapper class, you can process primitive data type values as objects. Java provides `Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer`, and `Long` wrapper classes in the `java.lang` package for primitive data types. The `Boolean` class wraps a Boolean value `true` or `false`. This section uses `Integer` and `Double` as examples to introduce the numeric wrapper classes.

naming convention

Note

Most wrapper class names for a primitive type are the same as the primitive data type name with the first letter capitalized. The exceptions are `Integer` for `int` and `Character` for `char`.

Numeric wrapper classes are very similar to each other. Each contains the methods `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`, `shortValue()`, and `byteValue()`. These methods "convert" objects into primitive-type values. The key features of `Integer` and `Double` are shown in Figure 10.14.

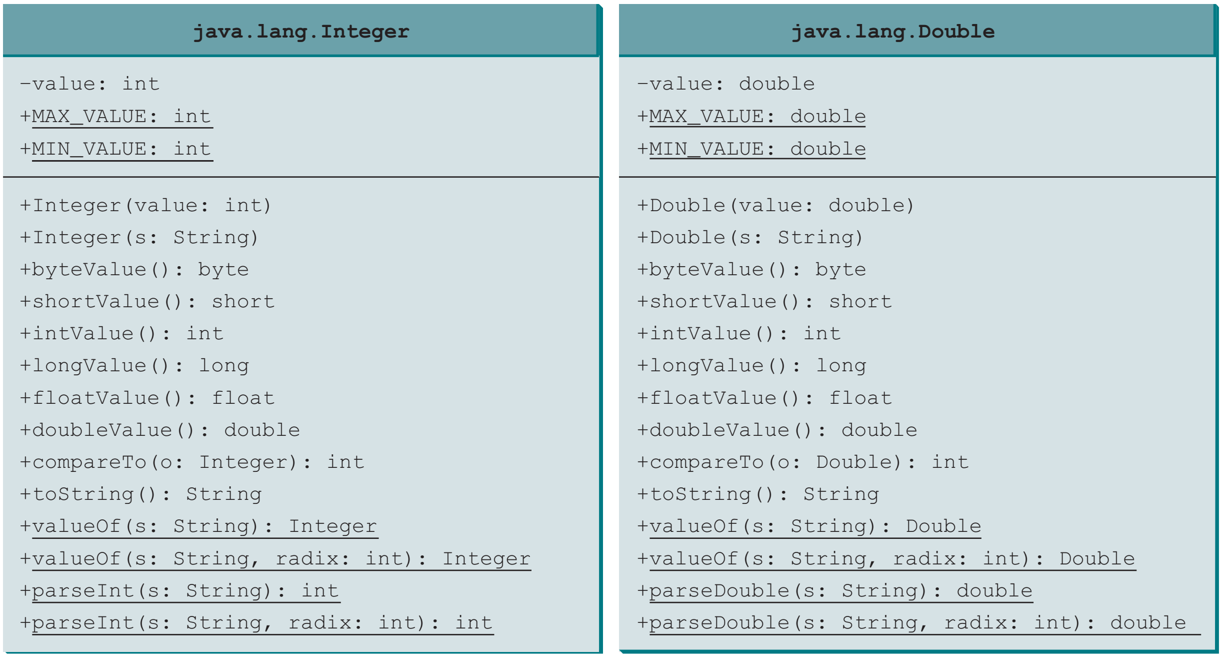

java.lang.Integer
-value: int
+MAX_VALUE: int
+MIN_VALUE: int
+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Integer): int
+toString(): String
+valueOf(s: String): Integer
+valueOf(s: String, radix: int): Integer
+parseInt(s: String): int
+parseInt(s: String, radix: int): int
java.lang.Double
-value: double
+MAX_VALUE: double
+MIN_VALUE: double
+Double(value: double)
+Double(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Double): int
+toString(): String
+valueOf(s: String): Double
+valueOf(s: String, radix: int): Double
+parseDouble(s: String): double
+parseDouble(s: String, radix: int): double

FIGURE 10.14 The wrapper classes provide constructors, constants, and conversion methods for manipulating various data types.

constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value—for example, `Double.valueOf(5.0)`, `Double.valueOf("5.0")`, `Integer.valueOf(5)`, and `Integer.valueOf("5")`.

no no-arg constructor

immutable

The wrapper classes do not have no-arg constructors. The instances of all wrapper classes are immutable; this means that, once the objects are created, their internal values cannot be changed.

The constructors in the wrapper classes are deprecated in Java 9. You should use the static `valueOf` method to create an instance. Java enables frequently used wrapper objects to be reused through the `valueOf` method. An instance created using `valueOf` maybe shared, which is fine because the wrapper objects are immutable. For example, in the following code, `x1` and `x2` are different objects, but `x3` and `x4` are the same objects created using the `valueOf` method. Note that `Integer x5 = 32`, is same as `Integer x5 = Integer.valueOf(32).`

```
Integer x1 = new Integer(“32”);
Integer x2 = new Integer(“32”);
Integer x3 = Integer.valueOf(“32”);
Integer x4 = Integer.valueOf(“32”);
Integer x5 = 32;
System.out.println(“x1 == x2 is” + (x1 == x2)); // Display false
System.out.println(“x1 == x3 is” + (x1 == x3)); // Display false
System.out.println(“x3 == x4 is” + (x3 == x4)); // Display true
System.out.println(“x3 == x5 is” + (x3 == x5)); // Display true
```

Note that the “frequently used wrapper objects are reused” in Java. Which ones are frequently used are not well defined in Java. In JDK 11, the frequently used wrapper objects are the byte-size integers between –128 and 127. For example, in the following code, `x1` and `x2` are not the same, although their `int` values are the same. However, it is preferred to use the `valueOf` method to create instances.

```
Integer x1 = Integer.valueOf("128");
Integer x2 = Integer.valueOf("128");
System.out.println("x1 == x2 is " + (x1 == x2)); // Display false
```

constants

Each numeric wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`. `MAX_VALUE` represents the maximum value of the corresponding primitive data type. For `Byte`, `Short`, `Integer`, and `Long`, `MIN_VALUE` represents the minimum `byte`, `short`, `int`, and `long` values. `Float` and `Double`, `MIN_VALUE` represents the minimum *positive* `float` and `double` values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E–45), and the maximum double floating-point number (1.79769313486231570e + 308d):

```
System.out.println("The maximum integer is " + Integer.MAX_VALUE);
System.out.println("The minimum positive float is " +
  Float.MIN_VALUE);
System.out.println(
  "The maximum double-precision floating-point number is " +
  Double.MAX_VALUE);
```

conversion methods

Each numeric wrapper class contains the methods `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`, and `shortValue()` for returning a `double`, `float`, `int`, `long`, or `short` value for the wrapper object. For example,

`Double.valueOf(12.4).intValue()` returns `12`;
`Integer.valueOf(12).doubleValue()` returns `12.0`;

`compareTo` method

Recall the `String` class contains the `compareTo` method for comparing two strings. The numeric wrapper classes contain the `compareTo` method for comparing two numbers and returns `1`, `0`, or `–1`, if this number is greater than, equal to, or less than the other number. For example,

`Double.valueOf(12.4).compareTo(Double.valueOf(12.3))` returns `1`;
`Double.valueOf(12.3).compareTo(Double.valueOf(12.3))` returns `0`;
`Double.valueOf(12.3).compareTo(Double.valueOf(12.51))` returns `–1`;

static `valueOf` methods

The numeric wrapper classes have a useful static method, `valueOf(String s)`. This method creates a new object initialized to the value represented by the specified string. For example,

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

static parsing methods

You have used the `parseInt` method in the `Integer` class to parse a numeric string into an `int` value and the `parseDouble` method in the `Double` class to parse a numeric string into a `double` value. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on `10` (decimal) or any specified radix (e.g., `2` for binary, `8` for octal, and `16` for hexadecimal).

```
// These two methods are in the Byte class
public static byte parseByte(String s)
public static byte parseByte(String s, int radix)

// These two methods are in the Short class
public static short parseShort(String s)
public static short parseShort(String s, int radix)

// These two methods are in the Integer class
public static int parseInt(String s)
public static int parseInt(String s, int radix)

// These two methods are in the Long class
public static long parseLong(String s)
public static long parseLong(String s, int radix)
```

```
// These two methods are in the Float class
public static float parseFloat(String s)
public static float parseFloat(String s, int radix)

// These two methods are in the Double class
public static double parseDouble(String s)
public static double parseDouble(String s, int radix)
```

For example,

`Integer.parseInt("11", 2)` returns `3`;
`Integer.parseInt("12", 8)` returns `10`;
`Integer.parseInt("13", 10)` returns `13`;
`Integer.parseInt("1A", 16)` returns `26`;

`Integer.parseInt("12", 2)` would raise a runtime exception because `12` is not a binary number.

converting decimal to hex

Note you can convert a decimal number into a hex number using the **`format`** method. For example,

`String.format("%x", 26)` returns `1A`;

Check Point

10.7.1 Describe primitive-type wrapper classes.

10.7.2 Can each of the following statements be compiled?

a. `Integer i = new Integer("23");`

b. `Integer i = new Integer(23);`

c. `Integer i = Integer.valueOf("23");`

d. `Integer i = Integer.parseInt("23", 8);`

e. `Double d = new Double();`

f. `Double d = Double.valueOf("23.45");`

g. `int i = (Integer.valueOf("23")).intValue();`

h. `double d = (Double.valueOf("23.4")).doubleValue();`

i. `int i = (Double.valueOf("23.4")).intValue();`

j. `String s = (Double.valueOf("23.4")).toString();`

10.7.3 How do you convert an integer into a string? How do you convert a numeric string into an integer? How do you convert a double number into a string? How do you convert a numeric string into a double value?

10.7.4 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    Integer x = Integer.valueOf(3);
    System.out.println(x.intValue());
    System.out.println(x.compareTo(4));
  }
}
```

10.7.5 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    System.out.println(Integer.parseInt("10"));
    System.out.println(Integer.parseInt("10", 10));
    System.out.println(Integer.parseInt("10", 16));
    System.out.println(Integer.parseInt("11"));
    System.out.println(Integer.parseInt("11", 10));
    System.out.println(Integer.parseInt("11", 16));
  }
}
```

10.8 Automatic Conversion between Primitive Types and Wrapper Class Types

Key Point

A primitive-type value can be automatically converted to an object using a wrapper class, and vice versa, depending on the context.

boxing
unboxing
autoboxing
autounboxing

Converting a primitive value to a wrapper object is called *boxing*. The reverse conversion is called *unboxing*. Java allows primitive types and wrapper classes to be converted automatically. The compiler will automatically box a primitive value that appears in a context requiring an object, and unbox an object that appears in a context requiring a primitive value. This is called *autoboxing* and *autounboxing*.

For instance, the following statement in (a) can be simplified as in (b) using autoboxing. The following statement in (a) is the same as in (b) due to autounboxing.

`Integer intObject = Integer.valueOf(2);` Equivalent `Integer intObject = 2;`

(a) autoboxing (b)

Consider the following example:

`int i = 1;` (a) and (b) are Equivalent `int i = Integer.valueOf(1);`

(a) (b)

```
1 Integer[] intArray = {1, 2, 3};
2 System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

In line 1, the primitive values **1**, **2**, and **3** are automatically boxed into objects **Integer.valueOf(1)**, **Integer.valueOf(2)**, and **Integer.valueOf(3)**. In line 2, the objects **intArray[0]**, **intArray[1]**, and **intArray[2]** are automatically unboxed into **int** values that are added together.

Check Point

10.8.1 What are autoboxing and autounboxing? Are the following statements correct?

```
a. Integer x = 3 + Integer.valueOf(5);
b. Integer x = 3;
c. Double x = 3;
d. Double x = 3.0;
e. int x = Integer.valueOf(3);
f. int x = Integer.valueOf(3) + Integer.valueOf(4);
```

10.8.2 Show the output of the following code.

```
public class Test {
  public static void main(String[] args) {
    Double x = 3.5;
    System.out.println(x.intValue());
    System.out.println(x.compareTo(4.5));
  }
}
```

10.9 The `BigInteger` and `BigDecimal` Classes

The `BigInteger` and `BigDecimal` classes can be used to represent integers or decimal numbers of any size and precision.

Key Point

If you need to compute with very large integers or high-precision floating-point values, you can use the `BigInteger` and `BigDecimal` classes in the `java.math` package. Both are *immutable*. The largest integer of the `long` type is `Long.MAX_VALUE` (i.e., `9223372036854775807`). An instance of `BigInteger` can represent an integer of any size. You can use `new BigInteger(String)` or `BigInteger.valueOf(long)` to create an instance of `BigInteger` and `new BigDecimal(String)` or `BigDecimal.valueOf(double)` to create an instance of `BigDecimal`, use the `add`, `subtract`, `multiply`, `divide`, and `remainder` methods to perform arithmetic operations, and use the `compareTo` method to compare two big numbers. For example, the following code creates two `BigInteger` objects and multiplies them:

immutable

VideoNote

Process large numbers

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

The output is `18446744073709551614`.

There is no limit to the precision of a `BigDecimal` object. The `divide` method may throw an `ArithmeticException` if the result cannot be terminated. However, you can use the overloaded `divide(BigDecimal d, int scale, RoundingMode roundingMode)` method to specify a scale and a rounding mode to avoid this exception, where `scale` is the maximum number of digits after the decimal point. For example, the following code creates two `BigDecimal` objects and performs division with scale `20` and rounding mode `RoundingMode.HALF_UP`:

```
BigDecimal a = new BigDecimal("1.0");
BigDecimal b = new BigDecimal("3");
BigDecimal c = a.divide(b, 20, RoundingMode.HALF_UP);
System.out.println(c);
```

The output is `0.33333333333333333334`.

Note

You can create a `BigDecimal` using either `new BigDecimal(String)` or `new BigDecimal(double)`. Since a double value is approximated, so the result from `new BigDecimal(double)` is unpredictable. For example, `new BigDecimal("1.0")` is not `1.0`, but is actually `0.1000000000000000055511151231257827021181583404541015625`. Therefore, it is a good idea to use `new BigDecimal(String)` to obtain a predictable `BigDecimal`.

`BigDecimal(String)` vs. `BigDecimal(double)`

Note the factorial of an integer can be very large. Listing 10.9 gives a method that can return the factorial of any integer.

LISTING 10.9 `LargeFactorial.java`

```
import java.util.Scanner;
import java.math.*;

public class LargeFactorial {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print(“Enter an integer: “);
    int n = input.nextInt();
```

constant

multiply

```
 9      System.out.println(n +"! is \n" + factorial(n));
10    }
11
12    public static BigInteger factorial(long n) {
13      BigInteger result = BigInteger.ONE;
14      for (int i = 1; i <= n; i++)
15        result = result.multiply(new BigInteger(i + “”));
16
17      return result;
18    }
19  }
```

```
Enter an integer: 50 ↵Enter
50! is
30414093201713378043612608166064768844377641568960512000000000000
```

`BigInteger.ONE` (line 13) is a constant defined in the `BigInteger` class. `BigInteger.ONE` is the same as `new BigInteger("1")`.

A new result is obtained by invoking the `multiply` method (line 15).

Check Point

10.9.1 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    java.math.BigInteger x = new java.math.BigInteger("3");
    java.math.BigInteger y = new java.math.BigInteger("7");
    java.math.BigInteger z = x.add(y);
    System.out.println("x is " + x);
    System.out.println("y is " + y);
    System.out.println("z is " + z);
  }
}
```

10.10 The String Class

Key Point

A `String` object is immutable; its contents cannot be changed once the string is created.

VideoNote

The `String` class

Strings were introduced in Section 4.4. You know strings are objects. You can invoke the `charAt(index)` method to obtain a character at the specified index from a string, the `length()` method to return the size of a string, the `substring` method to return a substring in a string, the `indexOf` and `lastIndexOf` methods to return the first or last index of a matching character or a substring, the `equals` and `compareTo` methods to compare two strings, and the `trim()` method to trim whitespace characters from the two ends of a string, and the `toLowerCase()` and `toUpperCase()` methods to return the lowercase and uppercase from a string. We will take a closer look at strings in this section.

The `String` class has 13 constructors and more than 40 methods for manipulating strings. Not only is it very useful in programming, but it is also a good example for learning classes and objects.

You can create a string object from a string literal or from an array of characters. To create a string from a string literal, use the syntax:

```
String newString = new String(stringLiteral);
```

The argument `stringLiteral` is a sequence of characters enclosed in double quotes. The following statement creates a `String` object `message` for the string literal `"Welcome to Java"`:

```
String message = new String("Welcome to Java");
```

Java treats a string literal as a `String` object. Thus, the following statement is valid:

string literal object

```
String message = "Welcome to Java";
```

You can also create a string from an array of characters. For example, the following statements create the string `"Good Day"`:

```
char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};
String message = new String(charArray);
```

Note

A `String` variable holds a reference to a `String` object that stores a string value. Strictly speaking, the terms `String` *variable*, `String` *object*, and *string value* are different, but most of the time the distinctions between them can be ignored. For simplicity, the term *string* will often be used to refer to `String` variable, `String` object, and string value.

`String` variable, `string` object, string value

10.10.1 Immutable Strings and Interned Strings

A `String` object is immutable; its contents cannot be changed. Does the following code change the contents of the string?

immutable

```
String s = "Java";
s = "HTML";
```

The answer is no. The first statement creates a `String` object with the content `"Java"` and assigns its reference to `s`. The second statement creates a new `String` object with the content `"HTML"` and assigns its reference to `s`. The first `String` object still exists after the assignment, but it can no longer be accessed, because variable `s` now points to the new object, as shown in Figure 10.15.

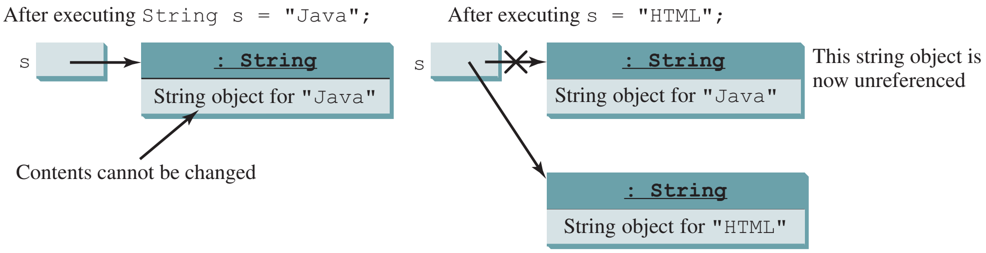
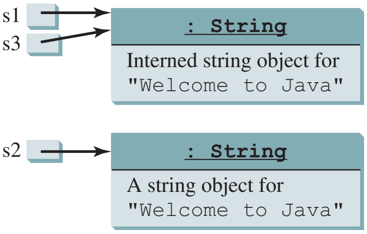

After executing String s = "Java";
s
: String
String object for "Java"
Contents cannot be changed
After executing s = "HTML";
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

FIGURE 10.15 Strings are immutable; once created, their contents cannot be changed.

Because strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called an *interned string*. For example, the following statements:

interned string

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

String s4 = new String("Welcome to Java");

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
System.out.println("s2 == s4 is " + (s2 == s4));
```


s1
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

display

```
s1 == s2 is false
s1 == s3 is true
s2 == s4 is false
```

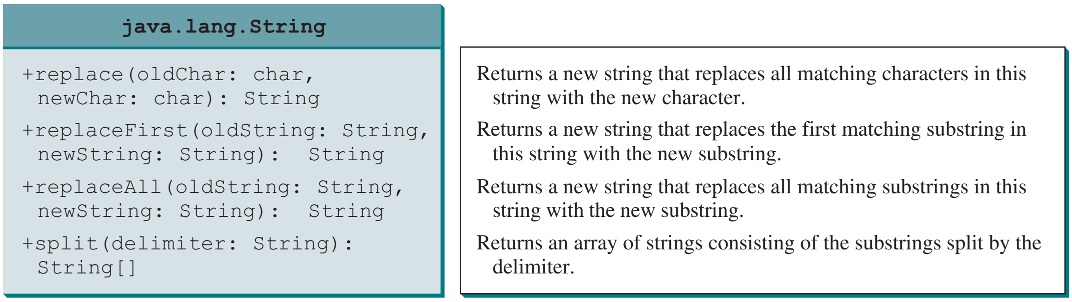
In the preceding statements, `s1` and `s3` refer to the same interned string—`"Welcome to Java"`—so `s1 == s3` is `true`. However, `s1 == s2` is `false`, because `s1` and `s2` are two different string objects, even though they have the same contents. `S2 == s4` is also `false`, because `s2` and `s4` are two different string objects.

Tip

You can create a `String` using `new String(stringLiteral)`. However, this is inefficient because it creates an unnecessary object. You should always simply use the `stringLiteral`. For example, use `String s = stringLiteral`; rather than `String s = new String(stringLiteral)`;

10.10.2 Replacing and Splitting Strings

The `String` class provides the methods for replacing and splitting strings, as shown in Figure 10.16.



java.lang.String

+replace(oldChar: char, newChar: char): String — Returns a new string that replaces all matching characters in this string with the new character.

+replaceFirst(oldString: String, newString: String): String — Returns a new string that replaces the first matching substring in this string with the new substring.

+replaceAll(oldString: String, newString: String): String — Returns a new string that replaces all matching substrings in this string with the new substring.

+split(delimiter: String): String[] — Returns an array of strings consisting of the substrings split by the delimiter.

FIGURE 10.16 The `String` class contains the methods for replacing and splitting strings.

Once a string is created, its contents cannot be changed. The methods `replace`, `replaceFirst`, and `replaceAll` return a new string derived from the original string (without changing the original string!). Several versions of the `replace` methods are provided to replace a character or a substring in the string with a new character or a new substring.

For example,

replace `"Welcome".replace('e', 'A')` returns a new string, `WAlcomA`.
replaceFirst `"Welcome".replaceFirst("e", "AB")` returns a new string, `WABlcome`.
replace `"Welcome".replace("e", "AB")` returns a new string, `WABlcomAB`.
replace `"Welcome".replace("el", "AB")` returns a new string, `WABcome`.
`"Welcome".replaceAll("e", "AB")` returns a new string, `WABlcomAB`.

replaceAll Note that `replaceAll(oldStr, newStr)` is the same as `replace(oldStr, newStr)` when used to replace all `oldStr` with `newStr`.

split The `split` method can be used to extract tokens from a string with the specified delimiters. For example, the following code

```
String[] tokens = "Java#HTML#Perl".split("#");
for (int i = 0; i < tokens.length; i++)
  System.out.print(tokens[i] + " ");
```

displays

```
Java HTML Perl
```

10.10.3 Matching, Replacing, and Splitting by Patterns

why regular expression?

Often you will need to write code that validates user input, such as to check whether the input is a number, a string with all lowercase letters, or a Social Security number. How do you write this type of code? A simple and effective way to accomplish this task is to use the regular expression.

regular expression
regex

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature.

`matches(regex)`

Let us begin with the `matches` method in the `String` class. At first glance, the `matches` method is very similar to the `equals` method. For example, the following two statements both evaluate to `true`:

```
"Java".matches("Java");
"Java".equals("Java");
```

However, the `matches` method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to `true`:

```
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

`Java.*` in the preceding statements is a regular expression. It describes a string pattern that begins with Java followed by *any* zero or more characters. Here, the substring matches any zero or more characters.

The following statement evaluates to `true`:

```
"440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}")
```

Here, `\\d` represents a single digit, and `\\d{3}` represents three digits.

The `replaceAll`, `replaceFirst`, and `split` methods can be used with a regular expression. For example, the following statement returns a new string that replaces `$`, `+`, or `#` in `a+b$#c` with the string `NNN`.

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

`replaceAll(regex)`

Here, the regular expression `[$+#]` specifies a pattern that matches `$`, `+`, or `#`. Thus, the output is `aNNNbNNNNNNc`.

The following statement splits the string into an array of strings delimited by punctuation marks.

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

for (int i = 0; i < tokens.length; i++)
  System.out.println(tokens[i]);
```

`split(regex)`

further studies

In this example, the regular expression `[.,:;?]` specifies a pattern that matches `.`, `,`, `:`, `;`, or `?`. Each of these characters is a delimiter for splitting the string. Thus, the string is split into `Java`, `C`, `C#`, and `C++`, which are stored in array `tokens`.

Regular expression patterns are complex for beginning students to understand. For this reason, simple patterns are introduced in this section. Please refer to Appendix H, Regular Expressions, to learn more about these patterns.

10.10.4 Conversion between Strings and Arrays

toCharArray

Strings are not arrays, but a string can be converted into an array and vice versa. To convert a string into an array of characters, use the **`toCharArray`** method. For example, the following statement converts the string **`Java`** to an array:

```
char[] chars = "Java".toCharArray();
```

Thus, **`chars[0]`** is **`J`**, **`chars[1]`** is **`a`**, **`chars[2]`** is **`v`**, and **`chars[3]`** is **`a`**.

You can also use the **`getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`** method to copy a substring of the string from index **`srcBegin`** to index **`srcEnd-1`** into a character array **`dst`** starting from index **`dstBegin`**. For example, the following code copies a substring **`"3720"`** in **`"CS3720"`** from index **`2`** to index **`6-1`** into the character array **`dst`** starting from index **`4`**:

getChars

```
char[] dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};
"CS3720".getChars(2, 6, dst, 4);
```

Thus, **`dst`** becomes **`{'J', 'A', 'V', 'A', '3', '7', '2', '0'}`**.

To convert an array of characters into a string, use the **`String(char[])`** constructor or the **`valueOf(char[])`** method. For example, the following statement constructs a string from an array using the **`String`** constructor:

```
String str = new String(new char[]{'J', 'a', 'v', 'a'});
```

valueOf

The next statement constructs a string from an array using the **`valueOf`** method.

```
String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});
```

10.10.5 Converting Characters and Numeric Values to Strings

Recall that you can use **`Double.parseDouble(str)`** or **`Integer.parseInt(str)`** to convert a string to a **`double`** value or an **`int`** value, and you can convert a character or a number into a string by using the string concatenating operator. Another way of converting a number into a string is to use the overloaded static **`valueOf`** method. This method can also be used to convert a character or an array of characters into a string, as shown in Figure 10.17.

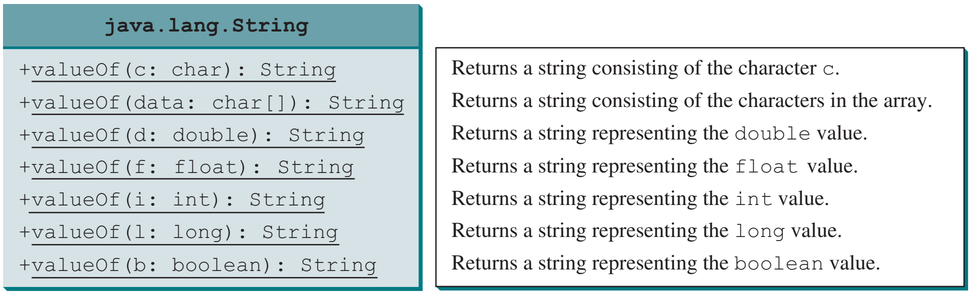

java.lang.String
+valueOf(c: char): String
+valueOf(data: char[]): String
+valueOf(d: double): String
+valueOf(f: float): String
+valueOf(i: int): String
+valueOf(l: long): String
+valueOf(b: boolean): String
Returns a string consisting of the character c.
Returns a string consisting of the characters in the array.
Returns a string representing the double value.
Returns a string representing the float value.
Returns a string representing the int value.
Returns a string representing the long value.
Returns a string representing the boolean value.

FIGURE 10.17 The **`String`** class contains the static methods for creating strings from primitive-type values.

For example, to convert a **`double`** value **`5.44`** to a string, use **`String.valueOf(5.44)`**. The return value is a string consisting of the characters **`'5'`**, **`'.'`**, **`'4'`**, and **`'4'`**.

10.10.6 Formatting Strings

The **`String`** class contains the static **`format`** method to return a formatted string. The syntax to invoke this method is

```
String.format(format, item1, item2, ..., itemk);
```

This method is similar to the `printf` method except that the `format` method returns a formatted string, whereas the `printf` method displays a formatted string. For example,

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
System.out.println(s);
```

displays

```
□□45.56□□□□14AB□□
```

where the square box (□) denotes a blank space.

Note

```
System.out.printf(format, item1, item2, ..., itemk);
```

is equivalent to

```
System.out.print(
  String.format(format, item1, item2, ..., itemk));
```

Check Point

10.10.1 Suppose `s1`, `s2`, `s3`, and `s4` are four strings, given as follows:

```
String s1 = "Welcome to Java";
String s2 = s1;
String s3 = new String("Welcome to Java");
String s4 = "Welcome to Java";
```

What are the results of the following expressions?

a. `s1 == s2`
b. `s1 == s3`
c. `s1 == s4`
d. `s1.equals(s3)`
e. `s1.equals(s4)`
f. `"Welcome to Java".replace("Java", "HTML")`
g. `s1.replace('o', 'T')`
h. `s1.replaceAll("o", "T")`
i. `s1.replaceFirst("o", "T")`
j. `s1.toCharArray()`

10.10.2 To create the string `Welcome to Java`, you may use a statement like this:

```
String s = "Welcome to Java";
```

or

```
String s = new String("Welcome to Java");
```

Which one is better? Why?

10.10.3 What is the output of the following code?

```
String s1 = "Welcome to Java";
String s2 = s1.replace("o", "abc");
System.out.println(s1);
System.out.println(s2);
```

10.10.4 Let `s1` be `" Welcome "` and `s2` be `" welcome "`. Write the code for the following statements:

a. Replace all occurrences of the character `e` with `E` in `s1` and assign the new string to `s3`.

b. Split `Welcome to Java and HTML` into an array `tokens` delimited by a space and assign the first two tokens into `s1` and `s2`.

10.10.5 Does any method in the `String` class change the contents of the string?

10.10.6 Suppose string `s` is created using `new String()`; what is `s.length()`?

10.10.7 How do you convert a `char`, an array of characters, or a number to a string?

10.10.8 Why does the following code cause a `NullPointerException`?

```java
public class Test {
  private String text;

  public Test(String s) {
    String text = s;
  }

  public static void main(String[] args) {
    Test test = new Test("ABC");
    System.out.println(test.text.toLowerCase());
  }
}
```

10.10.9 What is wrong in the following program?

```java
public class Test {
  String text;

  public void Test(String s) {
    text = s;
  }

  public static void main(String[] args) {
    Test test = new Test("ABC");
    System.out.println(test);
  }
}
```

10.10.10 Show the output of the following code:

```java
public class Test {
  public static void main(String[] args) {
    System.out.println("Hi, ABC, good".matches("ABC "));
    System.out.println("Hi, ABC, good".matches(".*ABC.*"));
    System.out.println("A,B;C".replaceAll(",;", "#"));
    System.out.println("A,B;C".replaceAll("[,;]", "#"));

    String[] tokens = "A,B;C".split("[,;]");
    for (int i = 0; i < tokens.length; i++)
      System.out.print(tokens[i] + " ");
  }
}
```

10.10.11 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    String s = "Hi, Good Morning";
    System.out.println(m(s));
  }

  public static int m(String s) {
    int count = 0;
    for (int i = 0; i < s.length(); i++)
      if (Character.isUpperCase(s.charAt(i)))
        count++;

    return count;
  }
}
```

10.11 The StringBuilder and StringBuffer Classes

The `StringBuilder` and `StringBuffer` classes are similar to the `String` class except that the `String` class is immutable.

Key Point

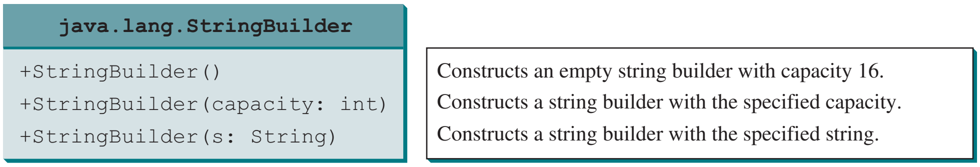
In general, the `StringBuilder` and `StringBuffer` classes can be used wherever a string is used. `StringBuilder` and `StringBuffer` are more flexible than `String`. You can add, insert, or append new contents into `StringBuilder` and `StringBuffer` objects, whereas the value of a `String` object is fixed once the string is created.

`StringBuilder`

The `StringBuilder` class is similar to `StringBuffer` except that the methods for modifying the buffer in `StringBuffer` are *synchronized*, which means that only one task is allowed to execute the methods. Use `StringBuffer` if the class might be accessed by multiple tasks concurrently, because synchronization is needed in this case to prevent corruptions to `StringBuffer`. Concurrent programming will be introduced in Chapter 32. Using `StringBuilder` is more efficient if it is accessed by just a single task, because no synchronization is needed in this case. The constructors and methods in `StringBuffer` and `StringBuilder` are almost the same. This section covers `StringBuilder`. You can replace `StringBuilder` in all occurrences in this section by `StringBuffer`. The program can compile and run without any other changes.

`StringBuilder` constructors

The `StringBuilder` class has three constructors and more than 30 methods for managing the builder and modifying strings in the builder. You can create an empty string builder using `new StringBuilder()` or a string builder from a string using `new StringBuilder(String)`, as shown in Figure 10.18.



```
java.lang.StringBuilder

+StringBuilder()                    Constructs an empty string builder with capacity 16.
+StringBuilder(capacity: int)       Constructs a string builder with the specified capacity.
+StringBuilder(s: String)           Constructs a string builder with the specified string.
```

FIGURE 10.18 The `StringBuilder` class contains the constructors for creating instances of `StringBuilder`.

10.11.1 Modifying Strings in the `StringBuilder`

You can append new contents at the end of a string builder, insert new contents at a specified position in a string builder, and delete or replace characters in a string builder, using the methods listed in Figure 10.19.

`java.lang.StringBuilder`	
`+append(data: char[]): StringBuilder`	Appends a `char` array into this string builder.
`+append(data: char[], offset: int, len: int): StringBuilder`	Appends a subarray in `data` into this string builder.
`+append(v: aPrimitiveType): StringBuilder`	Appends a primitive-type value as a string to this builder.
`+append(s: String): StringBuilder`	Appends a string to this string builder.
`+delete(startIndex: int, endIndex: int): StringBuilder`	Deletes characters from `startIndex` to `endIndex-1`.
`+deleteCharAt(index: int): StringBuilder`	Deletes a character at the specified index.
`+insert(index: int, data: char[], offset: int, len: int): StringBuilder`	Inserts a subarray of the data in the array into the builder at the specified index.
`+insert(offset: int, data: char[]): StringBuilder`	Inserts data into this builder at the position offset.
`+insert(offset: int, b: aPrimitiveType): StringBuilder`	Inserts a value converted to a string into this builder.
`+insert(offset: int, s: String): StringBuilder`	Inserts a string into this builder at the position offset.
`+replace(startIndex: int, endIndex: int, s: String): StringBuilder`	Replaces the characters in this builder from `startIndex` to `endIndex-1`with the specified string.
`+reverse(): StringBuilder`	Reverses the characters in the builder.
`+setCharAt(index: int, ch: char): void`	Sets a new character at the specified index in this builder.

FIGURE 10.19 The `StringBuilder` class contains the methods for modifying string builders.

The `StringBuilder` class provides several overloaded methods to append `boolean`, `char`, `char[]`, `double`, `float`, `int`, `long`, and `String` into a string builder. For example, the following code appends strings and characters into `stringBuilder` to form a new string, `Welcome to Java`:

append

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Welcome");
stringBuilder.append(' ');
stringBuilder.append("to");
stringBuilder.append(' ');
stringBuilder.append("Java");
```

The `StringBuilder` class also contains overloaded methods to insert `boolean`, `char`, `char array`, `double`, `float`, `int`, `long`, and `String` into a string builder. Consider the following code:

insert

```
stringBuilder.insert(11, "HTML and ");
```

Suppose `stringBuilder` contains `Welcome to Java` before the `insert` method is applied. This code inserts `"HTML and "` at position 11 in `stringBuilder` (just before the `J`). The new `stringBuilder` is `Welcome to HTML and Java`.

You can also delete characters from a string in the builder using the two `delete` methods, reverse the string using the `reverse` method, replace characters using the `replace` method, or set a new character in a string using the `setCharAt` method.

For example, suppose `stringBuilder` contains `Welcome to Java` before each of the following methods is applied:

delete `stringBuilder.delete(8, 11)` changes the builder to `Welcome Java`.
deleteCharAt `stringBuilder.deleteCharAt(8)` changes the builder to `Welcome o Java`.
reverse `stringBuilder.reverse()` changes the builder to `avaJ ot emocleW`.
replace `stringBuilder.replace(11, 15, "HTML")` changes the builder to `Welcome to HTML`.
setCharAt `stringBuilder.setCharAt(0, 'w')` sets the builder to `welcome to Java`.

All these modification methods except `setCharAt` do two things:

1. Change the contents of the string builder
2. Return the reference of the string builder

ignore return value

For example, the following statement:

```
StringBuilder stringBuilder1 = stringBuilder.reverse();
```

reverses the string in the builder and assigns the builder's reference to `stringBuilder1`. Thus, `stringBuilder` and `stringBuilder1` both point to the same `StringBuilder` object. Recall that a value-returning method can be invoked as a statement, if you are not interested in the return value of the method. In this case, the return value is simply ignored. For example, in the following statement:

```
stringBuilder.reverse();
```

the return value is ignored. Returning the reference of a `StringBuilder` enables the `StringBuilder` methods to be invoked in a chain such as the following:

```
stringBuilder.reverse().delete(8, 11).replace(11, 15, "HTML");
```

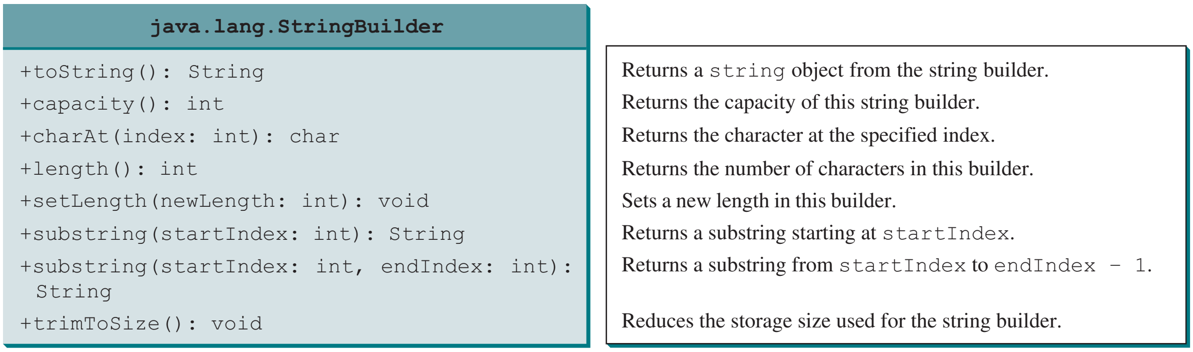
Tip

If a string does not require any change, use `String` rather than `StringBuilder`. `String` is more efficient than `StringBuilder`.

`String` or `StringBuilder`?

10.11.2 The `toString`, `capacity`, `length`, `setLength`, and `charAt` Methods

The `StringBuilder` class provides the additional methods for manipulating a string builder and obtaining its properties, as shown in Figure 10.20.



java.lang.StringBuilder	
+toString(): String	Returns a `string` object from the string builder.
+capacity(): int	Returns the capacity of this string builder.
+charAt(index: int): char	Returns the character at the specified index.
+length(): int	Returns the number of characters in this builder.
+setLength(newLength: int): void	Sets a new length in this builder.
+substring(startIndex: int): String	Returns a substring starting at `startIndex`.
+substring(startIndex: int, endIndex: int): String	Returns a substring from `startIndex` to `endIndex - 1`.
+trimToSize(): void	Reduces the storage size used for the string builder.

FIGURE 10.20 The `StringBuilder` class contains the methods for modifying string builders.

`capacity()`

The `capacity()` method returns the current capacity of the string builder. The capacity is the number of characters the string builder is able to store without having to increase its size.

`length()`
`setLength(int)`

The `length()` method returns the number of characters actually stored in the string builder. The `setLength(newLength)` method sets the length of the string builder. If the `newLength` argument is less than the current length of the string builder, the string builder is truncated to contain exactly the number of characters given by the `newLength` argument. If the `newLength` argument is greater than or equal to the current length, sufficient null characters (`\u0000`) are appended to the string builder so `length` becomes the `newLength` argument. The `newLength` argument must be greater than or equal to `0`.

`charAt(int)`

The **`charAt(index)`** method returns the character at a specific **`index`** in the string builder. The index is **`0`** based. The first character of a string builder is at index **`0`**, the next at index **`1`**, and so on. The **`index`** argument must be greater than or equal to **`0`**, and less than the length of the string builder.

length and capacity

Note

The length of the string builder is always less than or equal to the capacity of the builder. The length is the actual size of the string stored in the builder, and the capacity is the current size of the builder. The builder's capacity is automatically increased if more characters are added to exceed its capacity. Internally, a string builder is an array of characters, so the builder's capacity is the size of the array. If the builder's capacity is exceeded, the array is replaced by a new array. The new array size is **`2 * (the previous array size + 1)`**.

initial capacity

trimToSize()

Tip

You can use **`new StringBuilder(initialCapacity)`** to create a **`StringBuilder`** with a specified initial capacity. By carefully choosing the initial capacity, you can make your program more efficient. If the capacity is always larger than the actual length of the builder, the JVM will never need to reallocate memory for the builder. On the other hand, if the capacity is too large, you will waste memory space. You can use the **`trimToSize()`** method to reduce the capacity to the actual size.

10.11.3 Case Study: Ignoring Nonalphanumeric Characters When Checking Palindromes

Listing 5.14, Palindrome.java, considered all the characters in a string to check whether it is a palindrome. Write a new program that ignores nonalphanumeric characters in checking whether a string is a palindrome.

Here are the steps to solve the problem:

1. Filter the string by removing the nonalphanumeric characters. This can be done by creating an empty string builder, adding each alphanumeric character in the string to a string builder, and returning the string from the string builder. You can use the **`isLetterOrDigit(ch)`** method in the **`Character`** class to check whether character **`ch`** is a letter or a digit.
2. Obtain a new string that is the reversal of the filtered string. Compare the reversed string with the filtered string using the **`equals`** method.

The complete program is shown in Listing 10.10.

LISTING 10.10 `PalindromeIgnoreNonAlphanumeric.java`

```java
import java.util.Scanner;

public class PalindromeIgnoreNonAlphanumeric {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a string
    System.out.print("Enter a string: ");
    String s = input.nextLine();

    // Display result
14       System.out.println(“Ignoring nonalphanumeric characters, \nis “
15         + s + “ a palindrome? “ + isPalindrome(s));
16     }
17
18     /** Return true if a string is a palindrome */
19     public static boolean isPalindrome(String s) {
20       // Create a new string by eliminating nonalphanumeric chars
21       String s1 = filter(s);
22
23       // Create a new string that is the reversal of s1
24       String s2 = reverse(s1);
25
26       // Check if the reversal is the same as the original string
27       return s2.equals(s1);
28     }
29
30     /** Create a new string by eliminating nonalphanumeric chars */
31     public static String filter(String s) {
32       // Create a string builder
33       StringBuilder stringBuilder = new StringBuilder();
34
35       // Examine each char in the string to skip alphanumeric char
36       for (int i = 0; i < s.length(); i++) {
37         if (Character.isLetterOrDigit(s.charAt(i))) {
38           stringBuilder.append(s.charAt(i));
39         }
40       }
41
42       // Return a new filtered string
43       return stringBuilder.toString();
44     }
45
46     /** Create a new string by reversing a specified string */
47     public static String reverse(String s) {
48       StringBuilder stringBuilder = new StringBuilder(s);
49       stringBuilder.reverse(); // Invoke reverse in StringBuilder
50       return stringBuilder.toString();
51     }
52   }
```

check palindrome

add letter or digit

```
Enter a string: ab<c>cb?a ↵Enter
Ignoring nonalphanumeric characters,
is ab<c>cb?a a palindrome? true
```

```
Enter a string: abcc><?cab ↵Enter
Ignoring nonalphanumeric characters,
is abcc><?cab a palindrome? false
```

The **`filter(String s)`** method (lines 31–44) examines each character in string **`s`** and copies it to a string builder if the character is a letter or a numeric character. The **`filter`** method returns the string in the builder. The **`reverse(String s)`** method (lines 47–51) creates a new string that reverses the specified string **`s`**. The **`filter`** and **`reverse`** methods both return a new string. The original string is not changed.

The program in Listing 5.14 checks whether a string is a palindrome by comparing pairs of characters from both ends of the string. Listing 10.10 uses the `reverse` method in the `StringBuilder` class to reverse the string, then compares whether the two strings are equal to determine whether the original string is a palindrome.

Check Point

10.11.1 What is the difference between `StringBuilder` and `StringBuffer`?

10.11.2 How do you create a string builder from a string? How do you return a string from a string builder?

10.11.3 Write three statements to reverse a string `s` using the `reverse` method in the `StringBuilder` class.

10.11.4 Write three statements to delete a substring from a string `s` of `20` characters, starting at index `4` and ending with index `10`. Use the `delete` method in the `StringBuilder` class.

10.11.5 What is the internal storage for characters in a string and a string builder?

10.11.6 Suppose `s1` and `s2` are given as follows:

```java
StringBuilder s1 = new StringBuilder("Java");
StringBuilder s2 = new StringBuilder("HTML");
```

Show the value of `s1` after each of the following statements. Assume the statements are independent.

a. `s1.append(" is fun");`
b. `s1.append(s2);`
c. `s1.insert(2, "is fun");`
d. `s1.insert(1, s2);`
e. `s1.charAt(2);`
f. `s1.length();`
g. `s1.deleteCharAt(3);`
h. `s1.delete(1, 3);`
i. `s1.reverse();`
j. `s1.replace(1, 3, "Computer");`
k. `s1.substring(1, 3);`
l. `s1.substring(2);`

10.11.7 Show the output of the following program:

```java
public class Test {
  public static void main(String[] args) {
    String s = "Java";
    StringBuilder builder = new StringBuilder(s);
    change(s, builder);

    System.out.println(s);
    System.out.println(builder);
  }

  private static void change(String s, StringBuilder builder) {
    s = s + " and HTML";
    builder.append(" and HTML");
  }
}
```

KEY TERMS

abstract data type (ADT)	368
aggregation	376
boxing	386
class abstraction	368
class encapsulation	368
class's contract	368
composition	376
has-a relationship	376
multiplicity	375
stack	380
unboxing	386

CHAPTER SUMMARY

1. The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

2. Many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or wrap, a primitive data type into an object (e.g., wrapping `int` into the `Integer` class, and wrapping `double` into the `Double` class).

3. Java can automatically convert a primitive-type value to its corresponding wrapper object in the context and vice versa.

4. The `BigInteger` class is useful for computing and processing integers of any size. The `BigDecimal` class can be used to compute and process floating-point numbers with any arbitrary precision.

5. A `String` object is immutable; its contents cannot be changed. To improve efficiency and save memory, the JVM stores two literal strings that have the same character sequence in a unique object. This unique object is called an *interned string object*.

6. A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. You can match, replace, or split a string by specifying a pattern.

7. The `StringBuilder` and `StringBuffer` classes can be used to replace the `String` class. The `String` object is immutable, but you can add, insert, or append new contents into `StringBuilder` and `StringBuffer` objects. Use `String` if the string contents do not require any change and use `StringBuilder` or `StringBuffer` if they might change.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

Sections 10.2 and 10.3

***10.1** (*The `Time` class*) Design a class named `Time`. The class contains:

- The data fields `hour`, `minute`, and `second` that represent a time.
- A no-arg constructor that creates a `Time` object for the current time. (The values of the data fields will represent the current time.)

- A constructor that constructs a **`Time`** object with a specified elapsed time since midnight, January 1, 1970, in milliseconds. (The values of the data fields will represent this time.)
- A constructor that constructs a **`Time`** object with the specified hour, minute, and second.
- Three getter methods for the data fields **`hour`**, **`minute`**, and **`second`**, respectively.
- A method named **`setTime(long elapseTime)`** that sets a new time for the object using the elapsed time. For example, if the elapsed time is **`555550000`** milliseconds, the hour is **`10`**, the minute is **`19`**, and the second is **`10`**.

Draw the UML diagram for the class then implement the class. Write a test program that creates three **`Time`** objects (using **`new Time()`**, **`new Time(555550000)`**, and **`new Time(5, 23, 55)`**) and displays their hour, minute, and second in the format hour:minute:second.

(*Hint*: The first two constructors will extract the hour, minute, and second from the elapsed time. For the no-arg constructor, the current time can be obtained using **`System.currentTimeMillis()`**, as shown in Listing 2.7, ShowCurrentTime.java. Assume the time is in GMT.)

10.2 (*The* **`BMI`** *class*) Add the following new constructor in the **`BMI`** class:

```
/** Construct a BMI with the specified name, age, weight,
 * feet, and inches
 */
public BMI(String name, int age, double weight, double feet,
  double inches)
```

10.3 (*The* **`MyInteger`** *class*) Design a class named **`MyInteger`**. The class contains:

- An **`int`** data field named **`value`** that stores the **`int`** value represented by this object.
- A constructor that creates a **`MyInteger`** object for the specified **`int`** value.
- A getter method that returns the **`int`** value.
- The methods **`isEven()`**, **`isOdd()`**, and **`isPrime()`** that return **`true`** if the value in this object is even, odd, or prime, respectively.
- The static methods **`isEven(int)`**, **`isOdd(int)`**, and **`isPrime(int)`** that return **`true`** if the specified value is even, odd, or prime, respectively.
- The static methods **`isEven(MyInteger)`**, **`isOdd(MyInteger)`**, and **`isPrime(MyInteger)`** that return **`true`** if the specified value is even, odd, or prime, respectively.
- The methods **`equals(int)`** and **`equals(MyInteger)`** that return **`true`** if the value in this object is equal to the specified value.
- A static method **`parseInt(char[])`** that converts an array of numeric characters to an **`int`** value.
- A static method **`parseInt(String)`** that converts a string into an **`int`** value.

Draw the UML diagram for the class then implement the class. Write a client program that tests all methods in the class.

10.4 (*The* **`MyPoint`** *class*) Design a class named **`MyPoint`** to represent a point with **`x`**- and **`y`**-coordinates. The class contains:

- The data fields **`x`** and **`y`** that represent the coordinates with getter methods.
- A no-arg constructor that creates a point (**`0`**, **`0`**).
- A constructor that constructs a point with specified coordinates.
- A method named **`distance`** that returns the distance from this point to a specified point of the **`MyPoint`** type.

- A method named **`distance`** that returns the distance from this point to another point with specified **`x`**- and **`y`**-coordinates.
- A static method named distance that returns the distance from two MyPoint objects.

Draw the UML diagram for the class then implement the class. Write a test program that creates the two points (**`0`**, **`0`**) and (**`10`**, **`30.5`**) and displays the distance between them.

VideoNote
The `MyPoint` class

```
Enter an id: 4 ↵Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 ↵Enter
The balance is 100.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 2 ↵Enter
Enter an amount to withdraw: 3 ↵Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 ↵Enter
The balance is 97.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 3 ↵Enter
Enter an amount to deposit: 10 ↵Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 ↵Enter
The balance is 107.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 4 ↵Enter

Enter an id:
```

Sections 10.4–10.8

***10.5** (*Display the prime factors*) Write a program that prompts the user to enter a positive integer and displays all its smallest factors in decreasing order. For example, if the integer is `120`, the smallest factors are displayed as `5`, `3`, `2`, `2`, `2`. Use the `StackOfIntegers` class to store the factors (e.g., `2`, `2`, `2`, `3`, `5`) and retrieve and display them in reverse order.

***10.6** (*Display the prime numbers*) Write a program that displays all the prime numbers less than `120` in decreasing order. Use the `StackOfIntegers` class to store the prime numbers (e.g., `2`, `3`, `5`, . . .) and retrieve and display them in reverse order.

****10.7** (*Game: ATM machine*) Use the `Account` class created in Programming Exercise 9.7 to simulate an ATM machine. Create 10 accounts in an array with id `0`, `1`, . . . , `9`, and an initial balance of $100. The system prompts the user to enter an id. If the id is entered incorrectly, ask the user to enter a correct id. Once an id is accepted, the main menu is displayed as shown in the sample run. You can enter choice `1` for viewing the current balance, `2` for withdrawing money, `3` for depositing money, and `4` for exiting the main menu. Once you exit, the system will prompt for an id again. Thus, once the system starts, it will not stop.

*****10.8** (*Financial: the* `Tax` *class*) Programming Exercise 8.12 writes a program for computing taxes using arrays. Design a class named `Tax` to contain the following instance data fields:

- `int filingStatus`: One of the four tax-filing statuses: `0`—single filer, `1`—married filing jointly or qualifying widow(er), `2`—married filing separately, and `3`—head of household. Use the public static constants `SINGLE_FILER` (`0`), `MARRIED_JOINTLY_OR_QUALIFYING_WIDOW(ER)` (`1`), `MARRIED_SEPARATELY` (`2`), `HEAD_OF_HOUSEHOLD` (`3`) to represent the statuses.
- `int[][] brackets`: Stores the tax brackets for each filing status.
- `double[] rates`: Stores the tax rates for each bracket.
- `double taxableIncome`: Stores the taxable income.

Provide the getter and setter methods for each data field and the `getTax()` method that returns the tax. Also, provide a no-arg constructor and the constructor `Tax(filingStatus, brackets, rates, taxableIncome)`.

Draw the UML diagram for the class and then implement the class. Write a test program that uses the `Tax` class to print the 2001 and 2009 tax tables for taxable income from $50,000 to $60,000 with intervals of $1,000 for all four statuses. The tax rates for the year 2009 were given in Table 3.2. The tax rates for 2001 are shown in Table 10.1.

TABLE 10.1 2001 U.S. Federal Personal Tax Rates

Tax Rate	*Single Filers*	*Married—Filing Jointly or Qualifying Widow(er)*	*Married—Filing Separately*	*Head of Household*
15%	Up to $27,050	Up to $45,200	Up to $22,600	Up to $36,250
27.5%	$27,051–$65,550	$45,201–$109,250	$22,601–$54,625	$36,251–$93,650
30.5%	$65,551–$136,750	$109,251–$166,500	$54,626–$83,250	$93,651–$151,650
35.5%	$136,751–$297,350	$166,501–$297,350	$83,251–$148,675	$151,651–$297,350
39.1%	$297,351 or more	$297,351 or more	$ 148,676 or more	$297,351 or more

****10.9** (*The* `Course` *class*) Revise the `Course` class as follows:

- Revise the `getStudents()` method to return an array whose length is the same as the number of students in the course. (*Hint*: create a new array and copy students to it.)
- The array size is fixed in Listing 10.6. Revise the `addStudent` method to automatically increase the array size if there is no room to add more students. This is done by creating a new larger array and copying the contents of the current array to it.
- Implement the `dropStudent` method.
- Add a new method named `clear()` that removes all students from the course.

Test your program using https://liveexample.pearsoncmg.com/test/Exercise10_09.txt

***10.10** (*The* `Queue` *class*) Section 10.6 gives a class for `Stack`. Design a class named `Queue` for storing integers. Like a stack, a queue holds elements. In a stack, the elements are retrieved in a last-in first-out fashion. In a queue, the elements are retrieved in a first-in first-out fashion. The class contains:

- An `int[]` data field named `elements` that stores the `int` values in the queue.
- A data field named `size` that stores the number of elements in the queue.
- A constructor that creates a `Queue` object with default capacity `8`.
- The method `enqueue(int v)` that adds `v` into the queue.
- The method `dequeue()` that removes and returns the element from the queue.
- The method `empty()` that returns true if the queue is empty.
- The method `getSize()` that returns the size of the queue.

Draw an UML diagram for the class. Implement the class with the initial array size set to 8. The array size will be doubled once the number of the elements exceeds the size. After an element is removed from the beginning of the array, you need to shift all elements in the array one position to the left. Write a test program that adds 20 numbers from 1 to 20 into the queue then removes these numbers and displays them.

***10.11** (*Geometry: the* `Circle2D` *class*) Define the `Circle2D` class that contains:

- Two `double` data fields named `x` and `y` that specify the center of the circle with getter methods.
- A data field `radius` with a getter method.
- A no-arg constructor that creates a default circle with (`0`, `0`) for (`x`, `y`) and `1` for `radius`.
- A constructor that creates a circle with the specified `x`, `y`, and `radius`.
- A method `getArea()` that returns the area of the circle.
- A method `getPerimeter()` that returns the perimeter of the circle.
- A method `contains(double x, double y)` that returns `true` if the specified point (`x`, `y`) is inside this circle (see Figure 10.21a).
- A method `contains(Circle2D circle)` that returns `true` if the specified circle is inside this circle (see Figure 10.21b).
- A method `overlaps(Circle2D circle)` that returns `true` if the specified circle overlaps with this circle (see Figure 10.21c).

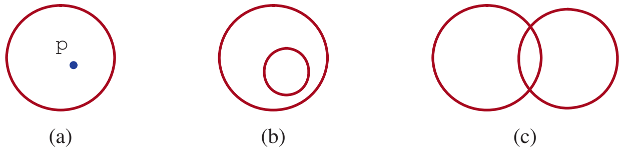


FIGURE 10.21 (a) A point is inside the circle. (b) A circle is inside another circle. (c) A circle overlaps another circle.

Draw the UML diagram for the class then implement the class. Write a test program that creates a `Circle2D` object `c1` (`new Circle2D(2, 2, 5.5)`), displays its area and perimeter, and displays the result of `c1.contains(3, 3)`, `c1.contains(new Circle2D(4, 5, 10.5))`, and `c1.overlaps(new Circle2D(3, 5, 2.3))`.

*****10.12** (*Geometry: the* `Triangle2D` *class*) Define the `Triangle2D` class that contains:

- Three points named `p1`, `p2`, and `p3` of the type `MyPoint` with getter and setter methods. `MyPoint` is defined in Programming Exercise 10.4.
- A no-arg constructor that creates a default triangle with the points (`0`, `0`), (`1`, `1`), and (`2`, `5`).
- A constructor that creates a triangle with the specified points.
- A method `getArea()` that returns the area of the triangle.
- A method `getPerimeter()` that returns the perimeter of the triangle.
- A method `contains(MyPoint p)` that returns `true` if the specified point `p` is inside this triangle (see Figure 10.22a).
- A method `contains(Triangle2D t)` that returns `true` if the specified triangle is inside this triangle (see Figure 10.22b).
- A method `overlaps(Triangle2D t)` that returns `true` if the specified triangle overlaps with this triangle (see Figure 10.22c).

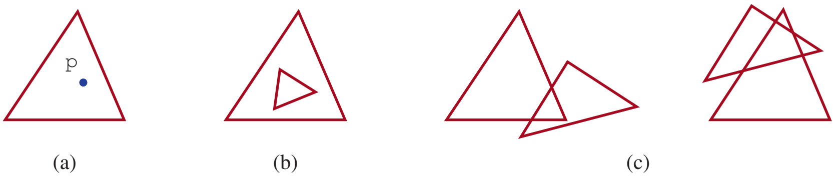


FIGURE 10.22 (a) A point is inside the triangle. (b) A triangle is inside another triangle. (c) A triangle overlaps another triangle.

Draw the UML diagram for the class and then implement the class. Write a test program that creates a `Triangle2D` object `t1` using the constructor `new Triangle2D(new MyPoint(2.5, 2), new MyPoint(4.2, 3), new MyPoint(5, 3.5))`, displays its area and perimeter, and displays the result of `t1.contains(3, 3)`, `r1.contains(new Triangle2D(new MyPoint(2.9, 2), new MyPoint(4, 1), MyPoint(1, 3.4)))`, and `t1.overlaps(new Triangle2D(new MyPoint(2, 5.5), new MyPoint(4, -3), MyPoint(2, 6.5)))`.

(*Hint*: For the formula to compute the area of a triangle, see Programming Exercise 2.19. To detect whether a point is inside a triangle, draw three dashed lines, as shown in Figure 10.23. Let Δ denote the area of a triangle. If ΔABp + ΔACp + ΔBCp == ΔABC, the point p is inside the triangle, as shown in Figure 10.23a. Otherwise, point p is not inside the triangle, as shown in Figure 10.23b.)

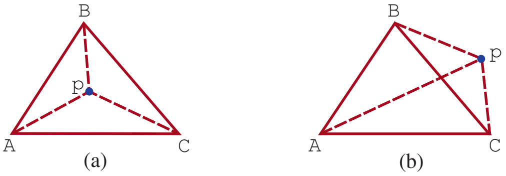
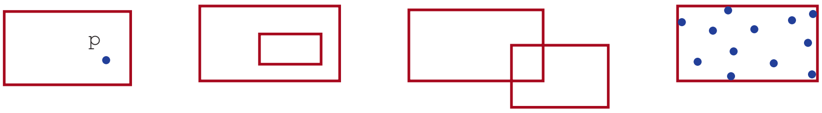


FIGURE 10.23 (a) A point is inside the triangle. (b) A point is outside the triangle.

*10.13 (*Geometry: the* `MyRectangle2D` *class*) Define the `MyRectangle2D` class that contains:

- Two `double` data fields named `x` and `y` that specify the center of the rectangle with getter and setter methods. (Assume the rectangle sides are parallel to `x-` or `y-`axis.)
- The data fields `width` and `height` with getter and setter methods.
- A no-arg constructor that creates a default rectangle with (`0`, `0`) for (`x`, `y`) and `1` for both `width` and `height`.
- A constructor that creates a rectangle with the specified `x`, `y`, `width`, and `height`.
- A method `getArea()` that returns the area of the rectangle.
- A method `getPerimeter()` that returns the perimeter of the rectangle.
- A method `contains(double x, double y)` that returns `true` if the specified point (`x`, `y`) is inside this rectangle (see Figure 10.24a).
- A method `contains(MyRectangle2D r)` that returns `true` if the specified rectangle is inside this rectangle (see Figure 10.24b).
- A method `overlaps(MyRectangle2D r)` that returns `true` if the specified rectangle overlaps with this rectangle (see Figure 10.24c).



(a) (b) (c) (d)

FIGURE 10.24 (a) A point is inside the rectangle. (b) A rectangle is inside another rectangle. (c) A rectangle overlaps another rectangle. (d) Points are enclosed inside a rectangle.

Draw the UML diagram for the class then implement the class. Write a test program that creates a `MyRectangle2D` object `r1` (`new MyRectangle2D(2, 2, 5.5, 4.9)`), displays its area and perimeter, and displays the result of `r1.contains(3, 3)`, `r1.contains(new MyRectangle2D(4, 5, 10.5, 3.2))`, and `r1.overlaps(new MyRectangle2D(3, 5, 2.3, 5.4))`.

*10.14 (*The* `MyDate` *class*) Design a class named `MyDate`. The class contains:

- The data fields `year`, `month`, and `day` that represent a date. `month` is 0-based, i.e., `0` is for January.
- A no-arg constructor that creates a `MyDate` object for the current date.
- A constructor that constructs a `MyDate` object with a specified elapsed time since midnight, January 1, 1970, in milliseconds.
- A constructor that constructs a `MyDate` object with the specified year, month, and day.
- Three getter methods for the data fields `year`, `month`, and `day`, respectively.
- A method named `setDate(long elapsedTime)` that sets a new date for the object using the elapsed time.

Draw the UML diagram for the class then implement the class. Write a test program that creates two `MyDate` objects (using `new MyDate()` and `new MyDate(34355555133101L)`) and displays their year, month, and day.

(*Hint*: The first two constructors will extract the year, month, and day from the elapsed time. For example, if the elapsed time is `561555550000` milliseconds, the year is `1987`, the month is `9`, and the day is `18`. You may use the `GregorianCalendar` class discussed in Programming Exercise 9.5 to simplify coding.)

***10.15** (*Geometry: the bounding rectangle*) A bounding rectangle is the minimum rectangle that encloses a set of points in a two-dimensional plane, as shown in Figure 10.24d. Write a method that returns a bounding rectangle for a set of points in a two-dimensional plane, as follows:

```
public static MyRectangle2D getRectangle(double[][] points)
```

The `Rectangle2D` class is defined in Programming Exercise 10.13. Write a test program that prompts the user to enter five points and displays the bounding rectangle's center, width, and height.

```
Enter five points: 1.0 2.5 3 4 5 6 7 8 9 10 ↵Enter
The bounding rectangle's center (5.0, 6.25), width 8.0, height 7.5
```

Section 10.9

***10.16** (*Divisible by* `2` *or* `3`) Find the first 10 numbers with `50` decimal digits that are divisible by `2` or `3`.

***10.17** (*Square numbers*) Find the first 10 square numbers that are greater than `Long.MAX_VALUE`. A square number is a number in the form of n^2. For example, 4, 9, and 16 are square numbers. Find an efficient approach to run your program fast.

***10.18** (*Large prime numbers*) Write a program that finds five prime numbers larger than `Long.MAX_VALUE`.

***10.19** (*Mersenne prime*) A prime number is called a *Mersenne prime* if it can be written in the form $2^p - 1$ for some positive integer p. Write a program that finds all Mersenne primes with $p \leq 100$ and displays the output as shown below. (*Hint*: You have to use `BigInteger` to store the number because it is too big to be stored in `long`. Your program may take several hours to run.)

```
    p         2^p - 1
---------------------

    2             3
    3             7
    5            31
...
```

***10.20** (*Approximate e*) Programming Exercise 5.26 approximates *e* using the following series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots + \frac{1}{i!}$$

In order to get better precision, use `BigDecimal` with `25` digits of precision in the computation. Write a program that displays the `e` value for `i = 100`, `200`, . . . , and `1000`.

10.21 (*Divisible by* `5` *or* `6`) Find the first 10 numbers greater than `Long.MAX_VALUE` that are divisible by `5` or `6`.

Sections 10.10 and 10.11

****10.22** (*Implement the* `String` *class*) The `String` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyString1`):

```
public MyString1(char[] chars);
public char charAt(int index);
public int length();
public MyString1 substring(int begin, int end);
public MyString1 toLowerCase();
public boolean equals(MyString1 s);
public static MyString1 valueOf(int i);
```

****10.23** (*Implement the* `String` *class*) The `String` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyString2`):

```
public MyString2(String s);
public int compare(String s);
public MyString2 substring(int begin);
public MyString2 toUpperCase();
public char[] toChars();
public static MyString2 valueOf(boolean b);
```

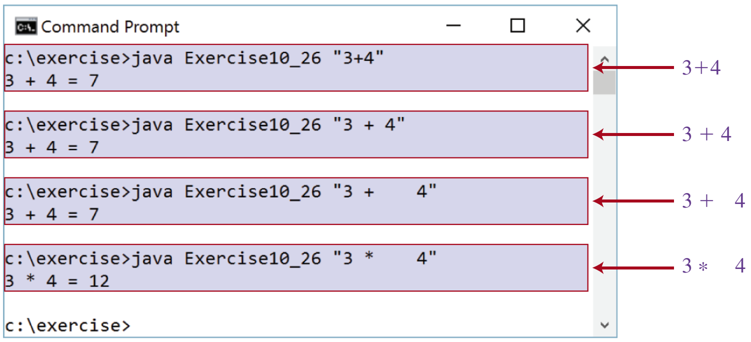
10.24 (*Implement the* `Character` *class*) The `Character` class is provided in the Java library. Provide your own implementation for this class. Name the new class `MyCharacter`.

****10.25** (*New string* `split` *method*) The `split` method in the `String` class returns an array of strings consisting of the substrings split by the delimiters. However, the delimiters are not returned. Implement the following new method that returns an array of strings consisting of the substrings split by the matching delimiters, including the matching delimiters.

```
public static String[] split(String s, String regex)
```

For example, `split("ab#12#453", "#")` returns `ab`, `#`, `12`, `#`, and `453` in an array of `String` and `split("a?b?gf#e", "[?#]")` returns `a`, `?`, `b`, `?`, `gf`, `#`, and `e` in an array of `String`.

***10.26** (*Calculator*) Revise Listing 7.9, Calculator.java, to accept an expression as a string in which the operands and operator are separated by zero or more spaces. For example, `3+4` and `3 + 4` are acceptable expressions. Here is a sample run:



```
c:\exercise>java Exercise10_26 "3+4"
3 + 4 = 7

c:\exercise>java Exercise10_26 "3 + 4"
3 + 4 = 7

c:\exercise>java Exercise10_26 "3 +    4"
3 + 4 = 7

c:\exercise>java Exercise10_26 "3 *    4"
3 * 4 = 12

c:\exercise>
```

****10.27** (*Implement the* **`StringBuilder`** *class*) The **`StringBuilder`** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **`MyStringBuilder1`**):

```
public MyStringBuilder1(String s);
public MyStringBuilder1 append(MyStringBuilder1 s);
public MyStringBuilder1 append(int i);
public int length();
public char charAt(int index);
public MyStringBuilder1 toLowerCase();
public MyStringBuilder1 substring(int begin, int end);
public String toString();
```

****10.28** (*Implement the* **`StringBuilder`** *class*) The **`StringBuilder`** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **`MyStringBuilder2`**):

```
public MyStringBuilder2();
public MyStringBuilder2(char[] chars);
public MyStringBuilder2(String s);
public MyStringBuilder2 insert(int offset, MyStringBuilder2 s);
public MyStringBuilder2 reverse();
public MyStringBuilder2 substring(int begin);
public MyStringBuilder2 toUpperCase();
```

CHAPTER

11

INHERITANCE AND POLYMORPHISM

Objectives

- To define a subclass from a superclass through inheritance (§11.2).
- To invoke the superclass's constructors and methods using the `super` keyword (§11.3).
- To override instance methods in the subclass (§11.4).
- To distinguish differences between overriding and overloading (§11.5).
- To explore the `toString()` method in the `Object` class (§11.6).
- To discover polymorphism and dynamic binding (§§11.7 and 11.8).
- To describe casting and explain why explicit downcasting is necessary (§11.9).
- To explore the `equals` method in the `Object` class (§11.10).
- To store, retrieve, and manipulate objects in an `ArrayList` (§11.11).
- To construct an array list from an array, to sort and shuffle a list, and to obtain max and min element from a list (§11.12).
- To implement a `Stack` class using `ArrayList` (§11.13).
- To enable data and methods in a superclass accessible from subclasses using the `protected` visibility modifier (§11.14).
- To prevent class extending and method overriding using the `final` modifier (§11.15).

11.1 Introduction

Key Point

Object-oriented programming allows you to define new classes from existing classes. This is called inheritance.

inheritance

As discussed in the preceding chapter, the procedural paradigm focuses on designing methods, and the object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

why inheritance?

Inheritance is an important and powerful feature for reusing software. Suppose you need to define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so as to avoid redundancy and make the system easy to comprehend and easy to maintain? The answer is to use inheritance.

11.2 Superclasses and Subclasses

Key Point

Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).

You use a class to model objects of the same type. Different classes may have some common properties and behaviors, which can be generalized in a class that can be shared by other classes. You can define a specialized class that extends the generalized class. The specialized classes inherit the properties and methods from the general class.

VideoNote
Geometric class hierarchy

Consider geometric objects. Suppose you want to design the classes to model geometric objects such as circles and rectangles. Geometric objects have many common properties and behaviors. They can be drawn in a certain color and be filled or unfilled. Thus, a general class `GeometricObject` can be used to model all geometric objects. This class contains the properties `color` and `filled` and their appropriate getter and setter methods. Assume this class also contains the `dateCreated` property, and the `getDateCreated()` and `toString()` methods. The `toString()` method returns a string representation of the object. Since a circle is a special type of geometric object, it shares common properties and methods with other geometric objects. Thus, it makes sense to define the `Circle` class that extends the `GeometricObject` class. Likewise, `Rectangle` can also be defined as a special type of `GeometricObject`. Figure 11.1 shows the relationship among these classes. A triangular arrow pointing to the generalized class is used to denote the inheritance relationship between the two classes involved.

subclass
superclass

In Java terminology, a class `C1` extended from another class `C2` is called a *subclass*, and `C2` is called a *superclass*. A superclass is also referred to as a *parent class* or a *base class*, and a subclass as a *child class*, an *extended class*, or a *derived class*. A subclass inherits accessible data fields and methods from its superclass and may also add new data fields and methods. Therefore, `Circle` and `Rectangle` are subclasses of `GeometricObject`, and `GeometricObject` is the superclass for `Circle` and `Rectangle`. A class defines a type. A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*. Therefore, you can say that `Circle` is a subtype of `GeometricObject`, and `GeometricObject` is a supertype for `Circle`.

subtype
supertype

is-a relationship

The subclass and its superclass are said to form a *is-a* relationship. A `Circle` object is a special type of general `GeometricObject`. The `Circle` class inherits all accessible data fields and methods from the `GeometricObject` class. In addition, it has a new data field, `radius`, and its associated getter and setter methods. The `Circle` class also contains the `getArea()`, `getPerimeter()`, and `getDiameter()` methods for returning the area, perimeter, and diameter of the circle.

width and height

The `Rectangle` class inherits all accessible data fields and methods from the `GeometricObject` class. In addition, it has the data fields `width` and `height` and their associated getter and setter methods. It also contains the `getArea()` and `getPerimeter()`

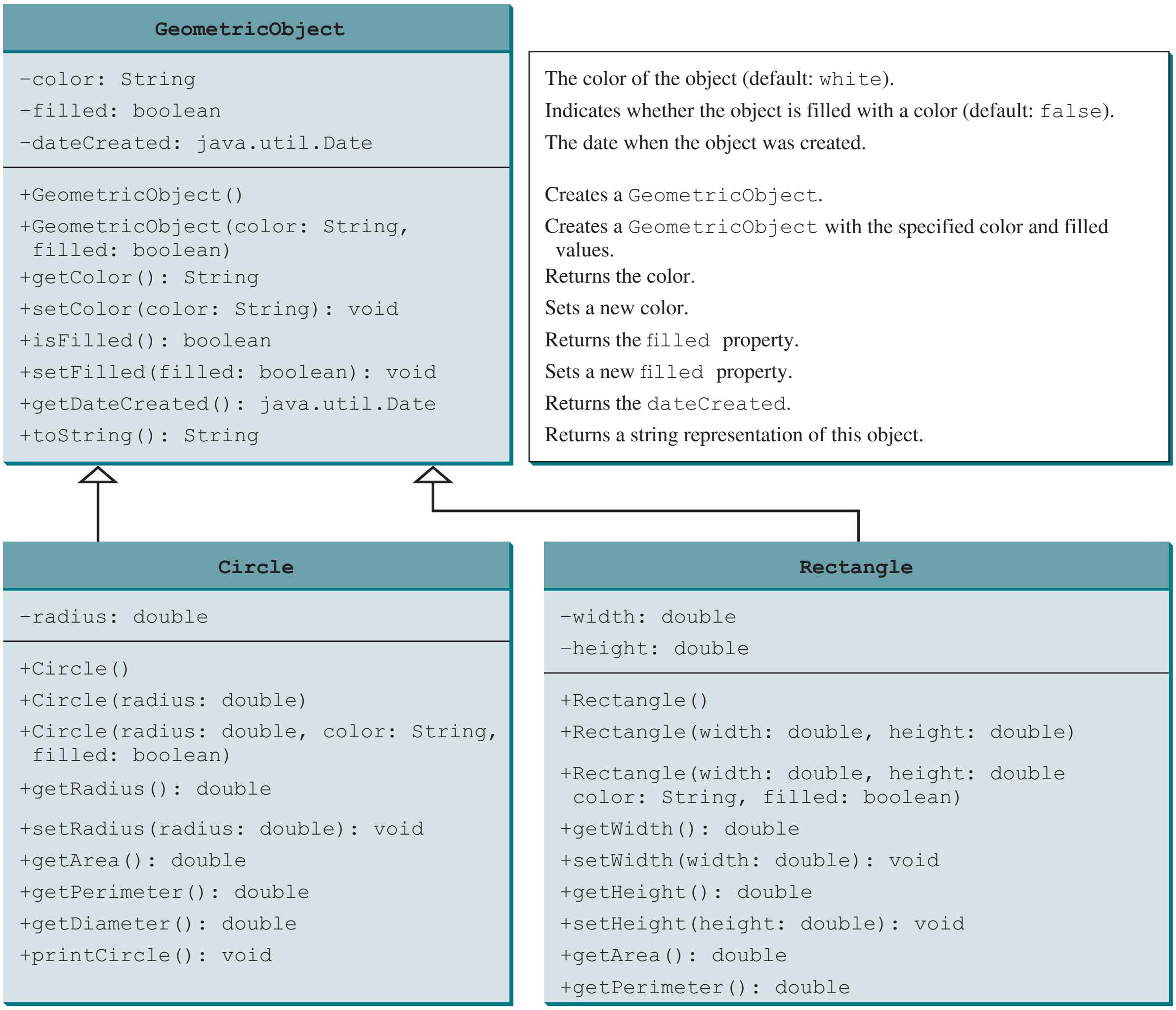

GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

FIGURE 11.1 The **`GeometricObject`** class is the superclass for **`Circle`** and **`Rectangle`**.

methods for returning the area and perimeter of the rectangle. Note that you may have used the terms width and length to describe the sides of a rectangle in geometry. The common terms used in computer science are width and height, where width refers to the horizontal length, and height to the vertical length.

The **`GeometricObject`**, **`Circle`**, and **`Rectangle`** classes are shown in Listings 11.1, 11.2, and 11.3 respectively.

LISTING 11.1 GeometricObject.java

```java
public class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /** Construct a default geometric object */
  public GeometricObject() {
    dateCreated = new java.util.Date();
  }

```

data fields

constructor

date constructed

```
  /** Construct a geometric object with the specified color
    *  and filled value */
  public GeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }

  /** Return color */
  public String getColor() {
    return color;
  }

  /** Set a new color */
  public void setColor(String color) {
    this.color = color;
  }

  /** Return filled. Since filled is boolean,
     its getter method is named isFilled */
  public boolean isFilled() {
    return filled;
  }

  /** Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }

  /** Get dateCreated */
  public java.util.Date getDateCreated() {
    return dateCreated;
  }

  /** Return a string representation of this object */
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }
}
```

LISTING 11.2 Circle.java

extends superclass

data fields

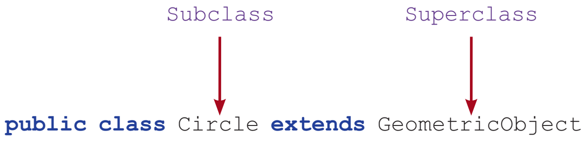
constructor

```
public class Circle extends GeometricObject {
  private double radius;

  public Circle() {
  }

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle(double radius,
      String color, boolean filled) {
    this.radius = radius;
    setColor(color);
    setFilled(filled);
  }

```

methods

```
18    /** Return radius */
19    public double getRadius() {
20      return radius;
21    }
22
23    /** Set a new radius */
24    public void setRadius(double radius) {
25      this.radius = radius;
26    }
27
28    /** Return area */
29    public double getArea() {
30      return radius * radius * Math.PI;
31    }
32
33    /** Return diameter */
34    public double getDiameter() {
35      return 2 * radius;
36    }
37
38    /** Return perimeter */
39    public double getPerimeter() {
40      return 2 * radius * Math.PI;
41    }
42
43    /** Print the circle info */
44    public void printCircle() {
45      System.out.println("The circle is created " + getDateCreated() +
46        " and the radius is " + radius);
47    }
48  }
```

The `Circle` class (Listing 11.2) extends the `GeometricObject` class (Listing 11.1) using the following syntax:



```
public class Circle extends GeometricObject
```

The keyword `extends` (lines 1 and 2) tells the compiler that the `Circle` class extends the `GeometricObject` class, thus inheriting the methods `getColor`, `setColor`, `isFilled`, `setFilled`, and `toString`.

The overloaded constructor `Circle(double radius, String color, boolean filled)` is implemented by invoking the `setColor` and `setFilled` methods to set the `color` and `filled` properties (lines 14 and 15). The public methods defined in the superclass `GeometricObject` are inherited in `Circle`, so they can be used in the `Circle` class.

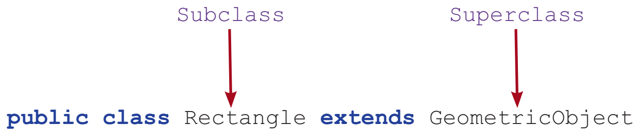
You might attempt to use the data fields `color` and `filled` directly in the constructor as follows:

private member in superclass

```
public Circle(double radius, String color, boolean filled) {
  this.radius = radius;
  this.color = color; // Illegal
  this.filled = filled; // Illegal
}
```

This is wrong because the private data fields `color` and `filled` in the `GeometricObject` class cannot be accessed in any class other than in the `GeometricObject` class itself. The only way to read and modify `color` and `filled` is through their getter and setter methods.

The `Rectangle` class (Listing 11.3) extends the `GeometricObject` class (Listing 11.1) using the following syntax:



```
public class Rectangle extends GeometricObject
```

The keyword `extends` (lines 1 and 2) tells the compiler the `Rectangle` class extends the `GeometricObject` class, thus inheriting the methods `getColor`, `setColor`, `isFilled`, `setFilled`, and `toString`.

LISTING 11.3 Rectangle.java

extends superclass
data fields
constructor
methods

```
 1 public class Rectangle extends GeometricObject {
 2   private double width;
 3   private double height;
 4 
 5   public Rectangle() {
 6   }
 7 
 8   public Rectangle(double width, double height) {
 9     this.width = width;
10     this.height = height;
11   }
12 
13   public Rectangle(
14       double width, double height, String color, boolean filled) {
15     this.width = width;
16     this.height = height;
17     setColor(color);
18     setFilled(filled);
19   }
20 
21   /** Return width */
22   public double getWidth() {
23     return width;
24   }
25 
26   /** Set a new width */
27   public void setWidth(double width) {
28     this.width = width;
29   }
30 
31   /** Return height */
32   public double getHeight() {
33     return height;
34   }
35 
36   /** Set a new height */
37   public void setHeight(double height) {
38     this.height = height;
39   }
40 
```

```
  /** Return area */
  public double getArea() {
    return width * height;
  }

  /** Return perimeter */
  public double getPerimeter() {
    return 2 * (width + height);
  }
}
```

The code in Listing 11.4 creates objects of **`Circle`** and **`Rectangle`** and invokes the methods on these objects. The **`toString()`** method is inherited from the **`GeometricObject`** class and is invoked from a **`Circle`** object (line 4) and a **`Rectangle`** object (line 11).

LISTING 11.4 `TestCircleRectangle.java`

```
 1  public class TestCircleRectangle {
 2    public static void main(String[] args) {
 3      Circle circle = new Circle(1);
 4      System.out.println("A circle " + circle.toString());
 5      System.out.println("The color is " + circle.getColor());
 6      System.out.println("The radius is " + circle.getRadius());
 7      System.out.println("The area is " + circle.getArea());
 8      System.out.println("The diameter is " + circle.getDiameter());
 9
10      Rectangle rectangle = new Rectangle(2, 4);
11      System.out.println("\nA rectangle " + rectangle.toString());
12      System.out.println("The area is " + rectangle.getArea());
13      System.out.println("The perimeter is " +
14        rectangle.getPerimeter());
15    }
16  }
```

`Circle` object
invoke `toString`
invoke `getColor`

`Rectangle` object
invoke `toString`

```
A circle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The color is white
The radius is 1.0
The area is 3.141592653589793
The diameter is 2.0
A rectangle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The area is 8.0
The perimeter is 12.0
```

Note the following points regarding inheritance:

- Contrary to the conventional interpretation, a subclass is not a subset of its superclass. In fact, a subclass usually contains more information and methods than its superclass.

more in subclass

- Private data fields in a superclass are not accessible outside the class. Therefore, they cannot be used directly in a subclass. They can, however, be accessed/mutated through public accessors/mutators if defined in the superclass.

private data fields

nonextensible is-a

- Not all is-a relationships should be modeled using inheritance. For example, a square is a rectangle, but you should not extend a `Square` class from a `Rectangle` class, because the `width` and `height` properties are not appropriate for a square. Instead, you should define a `Square` class to extend the `GeometricObject` class and define the `side` property for the side of a square.

no blind extension

- Inheritance is used to model the is-a relationship. Do not blindly extend a class just for the sake of reusing methods. For example, it makes no sense for a `Tree` class to extend a `Person` class, even though they share common properties such as `height` and `weight`. A subclass and its superclass must have the is-a relationship.

multiple inheritance

single inheritance

- Some programming languages allow you to derive a subclass from several classes. This capability is known as *multiple inheritance*. Java, however, does not allow multiple inheritance. A Java class may inherit directly from only one superclass. This restriction is known as *single inheritance*. If you use the `extends` keyword to define a subclass, it allows only one parent class. Nevertheless, multiple inheritance can be achieved through interfaces, which will be introduced in Section 13.5.

Check Point

11.2.1 True or false? A subclass is a subset of a superclass.

11.2.2 What keyword do you use to define a subclass?

11.2.3 What is single inheritance? What is multiple inheritance? Does Java support multiple inheritance?

11.3 Using the super Keyword

Key Point

The keyword `super` refers to the superclass and can be used to invoke the superclass's methods and constructors.

A subclass inherits accessible data fields and methods from its superclass. Does it inherit constructors? Can the superclass's constructors be invoked from a subclass? This section addresses these questions and their ramifications.

Section 9.14, The `this` Reference, introduced the use of the keyword `this` to reference the calling object. The keyword `super` refers to the superclass of the class in which `super` appears. It can be used in two ways:

1. To call a superclass constructor
2. To call a superclass method

11.3.1 Calling Superclass Constructors

A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited by a subclass. They can only be invoked from the constructors of the subclasses using the keyword `super`.

The syntax to call a superclass's constructor is:

```
super() or super(arguments);
```

The statement `super()` invokes the no-arg constructor of its superclass, and the statement `super(arguments)` invokes the superclass constructor that matches the `arguments`. The statement `super()` or `super(arguments)` must be the first statement of the subclass's constructor; this is the only way to explicitly invoke a superclass constructor. For example, the constructor in lines 11–16 in Listing 11.2 can be replaced by the following code:

```
public Circle(double radius, String color, boolean filled) {
  super(color, filled);
  this.radius = radius;
}
```

Caution

You must use the keyword `super` to call the superclass constructor, and the call must be the first statement in the constructor. Invoking a superclass constructor's name in a subclass causes a syntax error.

11.3.2 Constructor Chaining

A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts `super()` as the first statement in the constructor. For example:

```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

```
public ClassName(parameters) {
  // some statements
}
```

Equivalent

```
public ClassName(parameters) {
  super();
  // some statements
}
```

In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain. When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks. If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks. This process continues until the last constructor along the inheritance hierarchy is called. This is called *constructor chaining*.

constructor chaining

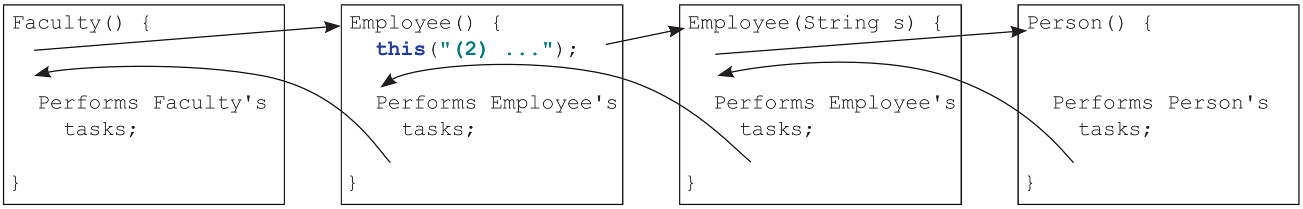
Consider the following code:

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Performs Faculty's tasks");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invokes Employee's overloaded constructor");
    System.out.println("(3) Performs Employee's tasks ");
}

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Performs Person's tasks");
  }
}
```

invoke overloaded constructor

```
(1) Performs Person's tasks
(2) Invokes Employee's overloaded constructor
(3) Performs Employee's tasks
(4) Performs Faculty's tasks
```

The program produces the preceding output. Why? Let us discuss the reason. In line 3, `new Faculty()` invokes `Faculty`'s no-arg constructor. Since `Faculty` is a subclass of `Employee`, `Employee`'s no-arg constructor is invoked before any statements in `Faculty`'s constructor are executed. `Employee`'s no-arg constructor invokes `Employee`'s second constructor (line 13). Since `Employee` is a subclass of `Person`, `Person`'s no-arg constructor is invoked before any statements in `Employee`'s second constructor are executed. This process is illustrated in the following figure.



no-arg constructor

Caution

If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors. Consider the following code:

```java
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

Since no constructor is explicitly defined in `Apple`, `Apple`'s default no-arg constructor is defined implicitly. Since `Apple` is a subclass of `Fruit`, `Apple`'s default constructor automatically invokes `Fruit`'s no-arg constructor. However, `Fruit` does not have a no-arg constructor, because `Fruit` has an explicit constructor defined. Therefore, the program cannot be compiled.

no-arg constructor

Design Guide

If possible, you should provide a no-arg constructor for every class to make the class easy to extend and to avoid errors.

11.3.3 Calling Superclass Methods

The keyword `super` can also be used to reference a method other than the constructor in the superclass. The syntax is

```java
super.method(arguments);
```

You could rewrite the `printCircle()` method in the `Circle` class as follows:

```java
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

It is not necessary to put `super` before `getDateCreated()` in this case, however, because `getDateCreated` is a method in the `GeometricObject` class and is inherited by the `Circle` class. Nevertheless, in some cases, as shown in the next section, the keyword `super` is needed.

Check Point

11.3.1 What is the output of running the class `C` in (a)? What problem arises in compiling the program in (b)?

```java
class A {
  public A() {
    System.out.println(
      "A's no-arg constructor is invoked");
  }
}

class B extends A {
}

public class C {
  public static void main(String[] args) {
    B b = new B();
  }
}
```

(a)

```java
class A {
  public A(int x) {
  }
}

class B extends A {
  public B() {
  }
}

public class C {
  public static void main(String[] args) {
    B b = new B();
  }
}
```

(b)

11.3.2 How does a subclass invoke its superclass's constructor?

11.3.3 True or false? When invoking a constructor from a subclass, its superclass's no-arg constructor is always invoked.

11.4 Overriding Methods

Key Point

To override a method, the method must be defined in the subclass using the same signature as in its superclass.

method overriding

A subclass inherits methods from a superclass. Sometimes, it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

The `toString` method in the `GeometricObject` class (lines 46–49 in Listing 11.1) returns the string representation of a geometric object. This method can be overridden to return the string representation of a circle. To override it, add the following new method in the `Circle` class in Listing 11.2:

```java
1  public class Circle extends GeometricObject {
2    // Other methods are omitted
3
4    // Override the toString method defined in the superclass
5    public String toString() {
6      return super.toString() + "\nradius is " + radius;
7    }
8  }
```

`toString` in superclass

The `toString()` method is defined in the `GeometricObject` class and modified in the `Circle` class. Both methods can be used in the `Circle` class. To invoke the `toString` method defined in the `GeometricObject` class from the `Circle` class, use `super.toString()` (line 6).

no super.super.methodName()

Can a subclass of `Circle` access the `toString` method defined in the `GeometricObject` class using syntax such as `super.super.toString()`? No. This is a syntax error.

Several points are worth noting:

override accessible instance method

- The overriding method must have the same signature as the overridden method and same or compatible return type. Compatible means that the overriding method's return type is a subtype of the overridden method's return type.
- An instance method can be overridden only if it is accessible. Thus, a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.
- Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden. The hidden static methods can be invoked using the syntax `SuperClassName.staticMethodName`.

cannot override static method

Check Point

11.4.1 True or false? You can override a private method defined in a superclass.

11.4.2 True or false? You can override a static method defined in a superclass.

11.4.3 How do you explicitly invoke a superclass's constructor from a subclass?

11.4.4 How do you invoke an overridden superclass method from a subclass?

11.5 Overriding vs. Overloading

Key Point

Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass.

You learned about overloading methods in Section 6.8. To override a method, the method must be defined in the subclass using the same signature and the same or compatible return type.

Let us use an example to show the differences between overriding and overloading. In (a) below, the method `p(double i)` in class `A` overrides the same method defined in class `B`. In (b), however, the class `A` has two overloaded methods: `p(double i)` and `p(int i)`. The method `p(double i)` is inherited from `B`.

```java
public class TestOverriding {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

(a)

```java
public class TestOverloading {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

(b)

When you run the `TestOverriding` class in (a), both `a.p(10)` and `a.p(10.0)` invoke the `p(double i)` method defined in class `A` to display `10.0`. When you run the `TestOverloading` class in (b), `a.p(10)` invokes the `p(int i)` method defined in class `A` to display `10` and `a.p(10.0)` invokes the `p(double i)` method defined in class `B` to display `20.0`.

Note the following:

- Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class, or in different classes related by inheritance.
- Overridden methods have the same signature; overloaded methods have the same name but different parameter lists.

To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place `@Override` before the overriding method in the subclass. For example,

override annotation

```
public class Circle extends GeometricObject {
  // Other methods are omitted

  @Override
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

`toString` in superclass

This annotation denotes that the annotated method is required to override a method in its superclass. If a method with this annotation does not override its superclass's method, the compiler will report an error. For example, if `toString` is mistyped as `tostring`, a compile error is reported. If the `@Override` annotation isn't used, the compiler won't report an error. Using the `@Override` annotation avoids mistakes.

Check Point

11.5.1 Identify the problems in the following code:

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    radius = radius;
  }

  public double getRadius() {
    return radius;
  }

  public double getArea() {
    return radius * radius * Math.PI;
  }
}

class B extends Circle {
  private double length;

  B(double radius, double length) {
    Circle(radius);
    length = length;
  }

  @Override
  public double getArea() {
    return getArea() * length;
  }
}
```

11.5.2 Explain the difference between method overloading and method overriding.

11.5.3 If a method in a subclass has the same signature as a method in its superclass with the same return type, is the method overridden or overloaded?

11.5.4 If a method in a subclass has the same signature as a method in its superclass with a different return type, will this be a problem?

11.5.5 If a method in a subclass has the same name as a method in its superclass with different parameter types, is the method overridden or overloaded?

11.5.6 What is the benefit of using the `@Override` annotation?

11.6 The `Object` Class and Its `toString()` Method

Key Point

Every class in Java is descended from the `java.lang.Object` class.

If no inheritance is specified when a class is defined, the superclass of the class is `Object` by default. For example, the following two class definitions in (a) and (b) are the same:

```
public class ClassName {
  ...
}
```

(a)

Equivalent

```
public class ClassName extends Object {
  ...
}
```

(b)

Classes such as `String`, `StringBuilder`, `Loan`, and `GeometricObject` are implicitly subclasses of `Object` (as are all the main classes you have seen in this book so far). It is important to be familiar with the methods provided by the `Object` class so that you can use them in your classes. This section introduces the `toString` method in the `Object` class.

toString()

The signature of the `toString()` method is:

```
public String toString()
```

string representation

Invoking `toString()` on an object returns a string that describes the object. By default, it returns a string consisting of a class name of which the object is an instance, an at sign (`@`), and the object's memory address in hexadecimal. For example, consider the following code for the `Loan` class defined in Listing 10.2:

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The output for this code displays something like `Loan@15037e5`. This message is not very helpful or informative. Usually you should override the `toString` method so that it returns a descriptive string representation of the object. For example, the `toString` method in the `Object` class was overridden in the `GeometricObject` class in lines 46–49 in Listing 11.1 as follows:

```
public String toString() {
  return "created on " + dateCreated + "\ncolor: " + color +
    " and filled: " + filled;
}
```

print object

Note

You can also pass an object to invoke `System.out.println(object)` or `System.out.print(object)`. This is equivalent to invoking `System.out.println(object.toString())` or `System.out.print(object.toString())`. Thus, you could replace `System.out.println(loan.toString())` with `System.out.println(loan)`.

11.7 Polymorphism

Polymorphism means that a variable of a supertype can refer to a subtype object.

Key Point

The three pillars of object-oriented programming are encapsulation, inheritance, and polymorphism. You have already learned the first two. This section introduces polymorphism.

The inheritance relationship enables a subclass to inherit features from its superclass with additional new features. A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa. For example, every circle is a geometric object, but not every geometric object is a circle. Therefore, you can always pass an instance of a subclass to a parameter of its superclass type. Consider the code in Listing 11.5.

Listing 11.5 `PolymorphismDemo.java`

```
 1  public class PolymorphismDemo {
 2    /** Main method */
 3    public static void main(String[] args) {
 4      // Display circle and rectangle properties
 5      displayObject(new Circle(1, "red", false));
 6      displayObject(new Rectangle(1, 1, "black", true));
 7    }
 8
 9    /** Display geometric object properties */
10    public static void displayObject(GeometricObject object) {
11      System.out.println("Created on " + object.getDateCreated() +
12        ". Color is " + object.getColor());
13    }
14  }
```

polymorphic call
polymorphic call

```
Created on Mon Mar 09 19:25:20 EDT 2011. Color is red
Created on Mon Mar 09 19:25:20 EDT 2011. Color is black
```

The method `displayObject` (line 10) takes a parameter of the `GeometricObject` type. You can invoke `displayObject` by passing any instance of `GeometricObject` (e.g., `new Circle(1, "red", false)` and `new Rectangle(1, 1, "black", true)` in lines 5 and 6). An object of a subclass can be used wherever its superclass object is used. This is commonly known as *polymorphism* (from a Greek word meaning "many forms"). In simple terms, polymorphism means that a variable of a supertype can refer to a subtype object.

what is polymorphism?

11.7.1 What are the three pillars of object-oriented programming? What is polymorphism?

Check Point

11.8 Dynamic Binding

A method can be implemented in several classes along the inheritance chain. The JVM decides which method is invoked at runtime.

Key Point

A method can be defined in a superclass and overridden in its subclass. For example, the `toString()` method is defined in the `Object` class and overridden in `GeometricObject`. Consider the following code:

```
Object o = new GeometricObject();
System.out.println(o.toString());
```

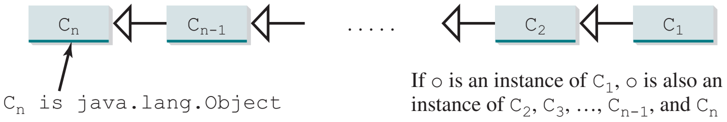
declared type

actual type

dynamic binding

Which `toString()` method is invoked by `o`? To answer this question, we first introduce two terms: declared type and actual type. A variable must be declared a type. The type that declares a variable is called the variable's *declared type*. Here, `o`'s declared type is `Object`. A variable of a reference type can hold a `null` value or a reference to an instance of the declared type. The instance may be created using the constructor of the declared type or its subtype. The *actual type* of the variable is the actual class for the object referenced by the variable at runtime. Here, `o`'s actual type is `GeometricObject`, because `o` references an object created using `new GeometricObject()`. Which `toString()` method is invoked by `o` is determined by `o`'s actual type. This is known as *dynamic binding*.

Dynamic binding works as follows: Suppose that an object `o` is an instance of classes C_1, C_2, . . . , C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, . . . , and C_{n-1} is a subclass of C_n, as shown in Figure 11.2. That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the `Object` class. If `o` invokes a method `p`, the JVM searches for the implementation of the method `p` in C_1, C_2, . . . , C_{n-1}, and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



C_n
C_{n-1}
. . . . .
C_2
C_1
C_n is java.lang.Object
If o is an instance of C_1, o is also an instance of C_2, C_3, …, C_{n-1}, and C_n

FIGURE 11.2 The method to be invoked is dynamically bound at runtime.

VideoNote
Polymorphism and dynamic binding demo

Listing 11.6 gives an example to demonstrate dynamic binding.

LISTING 11.6 DynamicBindingDemo.java

polymorphic call

dynamic binding

override toString()

override toString()

```
public class DynamicBindingDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  @Override
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  @Override
26    public String toString() {
27      return "Person";
28    }
29  }
```

```
Student
Student
Person
java.lang.Object@130c19b
```

Method `m` (line 9) takes a parameter of the `Object` type. You can invoke `m` with any object (e.g., `new GraduateStudent()`, `new Student()`, `new Person()`, and `new Object()`) in lines 3–6).

When the method `m(Object x)` is executed, the argument `x`’s `toString` method is invoked. `x` may be an instance of `GraduateStudent`, `Student`, `Person`, or `Object`. The `toString` method is implemented in `Student`, `Person`, and `Object`. Which implementation is used will be determined by `x`’s actual type at runtime. Invoking `m(new GraduateStudent())` (line 3) causes the `toString` method defined in the `Student` class to be invoked.

Invoking `m(new Student())` (line 4) causes the `toString` method defined in the `Student` class to be invoked; invoking `m(new Person())` (line 5) causes the `toString` method defined in the `Person` class to be invoked; and invoking `m(new Object())` (line 6) causes the `toString` method defined in the `Object` class to be invoked.

matching vs. binding

Matching a method signature and binding a method implementation are two separate issues. The *declared type* of the reference variable decides which method to match at compile time. The compiler finds a matching method according to the parameter type, number of parameters, and order of the parameters at compile time. A method may be implemented in several classes along the inheritance chain. The JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable.

Check Point

11.8.1 What is dynamic binding?

11.8.2 Describe the difference between method matching and method binding.

11.8.3 Can you assign `new int[50]`, `new Integer[50]`, `new String[50]`, or `new Object[50]` into a variable of `Object[]` type?

11.8.4 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    Integer[] list1 = {12, 24, 55, 1};
    Double[] list2 = {12.4, 24.0, 55.2, 1.0};
    int[] list3 = {1, 2, 3};
    printArray(list1);
    printArray(list2);
    printArray(list3);
  }

  public static void printArray(Object[] list) {
    for (Object o: list)
      System.out.print(o + " ");
    System.out.println();
  }
}
```

11.8.5 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    new Person().printPerson();
    new Student().printPerson();
  }
}

class Student extends Person {
  @Override
  public String getInfo() {
    return "Student";
  }
}

class Person {
  public String getInfo() {
    return "Person";
  }

  public void printPerson() {
    System.out.println(getInfo());
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    new Person().printPerson();
    new Student().printPerson();
  }
}

class Student extends Person {
  private String getInfo() {
    return "Student";
  }
}

class Person {
  private String getInfo() {
    return "Person";
  }

  public void printPerson() {
    System.out.println(getInfo());
  }
}
```

(b)

11.8.6 Show the output of following program:

```
public class Test {
  public static void main(String[] args) {
    A a = new A(3);
  }
}

class A extends B {
  public A(int t) {
    System.out.println("A's constructor is invoked");
  }
}

class B {
  public B() {
    System.out.println("B's constructor is invoked");
  }
}
```

Is the no-arg constructor of `Object` invoked when `new A(3)` is invoked?

11.8.7 Show the output of following program:

```
public class Test {
  public static void main(String[] args) {
    new A();
    new B();
  }
}
```

```
class A {
  int i = 7;

  public A() {
    setI(20);
    System.out.println("i from A is " + i);
  }

  public void setI(int i) {
    this.i = 2 * i;
  }
}

class B extends A {
  public B() {
    System.out.println("i from B is " + i);
  }

  public void setI(int i) {
    this.i = 3 * i;
  }
}
```

11.9 Casting Objects and the `instanceof` Operator

One object reference can be typecast into another object reference. This is called casting object.

Key Point

In the preceding section, the statement

casting object

```
m(new Student());
```

assigns the object `new Student()` to a parameter of the `Object` type. This statement is equivalent to

```
Object o = new Student(); // Implicit casting
m(o);
```

implicit casting

The statement `Object o = new Student()`, known as *implicit casting*, is legal because an instance of `Student` is an instance of `Object`.

Suppose you want to assign the object reference `o` to a variable of the `Student` type using the following statement:

```
Student b = o;
```

explicit casting

In this case a compile error would occur. Why does the statement `Object o = new Student()` work, but `Student b = o` doesn't? The reason is that a `Student` object is always an instance of `Object`, but an `Object` is not necessarily an instance of `Student`. Even though you can see that `o` is really a `Student` object, the compiler is not clever enough to know it. To tell the compiler `o` is a `Student` object, use *explicit casting*. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o; // Explicit casting
```

upcasting
downcasting

It is always possible to cast an instance of a subclass to a variable of a superclass (known as *upcasting*) because an instance of a subclass is *always* an instance of its superclass. When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used to confirm your intention to the compiler with the `(SubclassName)` cast notation. For the casting to be successful, you must make sure the object to be cast is an instance of the subclass. If the superclass object is not an instance of the subclass, a runtime ***`ClassCastException`*** occurs. For example, if an object is not an instance of `Student`, it cannot be cast into a variable of `Student`. It is a good practice, therefore, to ensure the object is an instance of another object before attempting a casting. This can be accomplished by using the ***`instanceof`*** operator. Consider the following code:

`ClassCastException`

`instanceof`

```
void someMethod(Object myObject) {
  ... // Some lines of code
  /** Perform casting if myObject is an instance of Circle */
  if (myObject instanceof Circle) {
    System.out.println("The circle diameter is " +
      ((Circle)myObject).getDiameter());
    ...
  }
}
```

You may be wondering why casting is necessary. The variable `myObject` is declared `Object`. The *declared type* decides which method to match at compile time. Using `myObject.getDiameter()` would cause a compile error, because the `Object` class does not have the `getDiameter` method. The compiler cannot find a match for `myObject.getDiameter()`. Therefore, it is necessary to cast `myObject` into the `Circle` type to tell the compiler that `myObject` is also an instance of `Circle`.

Why not declare `myObject` as a `Circle` type in the first place? To enable generic programming, it is a good practice to declare a variable with a supertype that can accept an object of any subtype.

lowercase keywords

Note

`instanceof` is a Java keyword. Every letter in a Java keyword is in lowercase.

casting analogy

Tip

To help understand casting, you may also consider the analogy of fruit, apple, and orange, with the `Fruit` class as the superclass for `Apple` and `Orange`. An apple is a fruit, so you can always safely assign an instance of `Apple` to a variable for `Fruit`. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of `Fruit` to a variable of `Apple`.

Listing 11.7 demonstrates polymorphism and casting. The program creates two objects (lines 5 and 6), a `circle` and a `rectangle`, and invokes the `displayObject` method to display them (lines 9 and 10). The `displayObject` method displays the area and diameter if the object is a circle (line 15), and the area if the object is a rectangle (line 21).

LISTING 11.7 CastingDemo.java

```
 1  public class CastingDemo {
 2    /** Main method */
 3    public static void main(String[] args) {
 4      // Create and initialize two objects
 5      Object object1 = new Circle(1);
 6      Object object2 = new Rectangle(1, 1);
 7
 8      // Display circle and rectangle
 9      displayObject(object1);
10      displayObject(object2);
11    }
12
13    /** A method for displaying an object */
14    public static void displayObject(Object object) {
15      if (object instanceof Circle) {
16        System.out.println("The circle area is " +
17          ((Circle)object).getArea());
18        System.out.println("The circle diameter is " +
19          ((Circle)object).getDiameter());
20      }
21      else if (object instanceof Rectangle) {
22        System.out.println("The rectangle area is " +
23          ((Rectangle)object).getArea());
24      }
25    }
26  }
```

polymorphic call

polymorphic call

```
The circle area is 3.141592653589793
The circle diameter is 2.0
The rectangle area is 1.0
```

The `displayObject(Object object)` method is an example of generic programming. It can be invoked by passing any instance of `Object`.

The program uses implicit casting to assign a `Circle` object to `object1` and a `Rectangle` object to `object2` (lines 5 and 6), then invokes the `displayObject` method to display the information on these objects (lines 9–10).

In the `displayObject` method (lines 14–25), explicit casting is used to cast the object to `Circle` if the object is an instance of `Circle`, and the methods `getArea` and `getDiameter` are used to display the area and diameter of the circle.

Casting can be done only when the source object is an instance of the target class. The program uses the `instanceof` operator to ensure that the source object is an instance of the target class before performing a casting (line 15).

Explicit casting to `Circle` (lines 17 and 19) and to `Rectangle` (line 23) is necessary because the `getArea` and `getDiameter` methods are not available in the `Object` class.

Caution

The object member access operator (`.`) has higher precedence than the casting operator. Use parentheses to ensure that casting is done before the `.` operator, as in

precedes casting

```
((Circle)object).getArea();
```

Casting a primitive-type value is different from casting an object reference. Casting a primitive-type value returns a new value. For example:

```
int age = 45;
byte newAge = (byte)age; // A new value is assigned to newAge
```

However, casting an object reference does not create a new object. For example:

```
Object o = new Circle();
Circle c = (Circle)o; // No new object is created
```

Now, reference variables `o` and `c` point to the same object.

Check Point

11.9.1 Indicate true or false for the following statements:

a. You can always successfully cast an instance of a subclass to a superclass.

b. You can always successfully cast an instance of a superclass to a subclass.

11.9.2 For the `GeometricObject` and `Circle` classes in Listings 11.1 and 11.2, answer the following questions:

a. Assume that `circle` and `object1` are created as follows:

```
Circle circle = new Circle(1);
GeometricObject object1 = new GeometricObject();
```

Are the following Boolean expressions true or false?

```
(circle instanceof GeometricObject)
(object instanceof GeometricObject)
(circle instanceof Circle)
(object instanceof Circle)
```

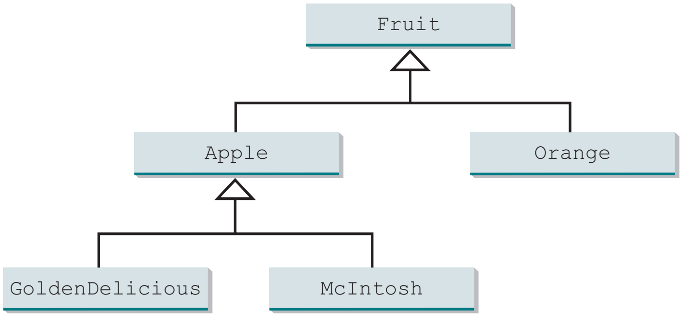
b. Can the following statements be compiled?

```
Circle circle = new Circle(5);
GeometricObject object = circle;
```

c. Can the following statements be compiled?

```
GeometricObject object = new GeometricObject();
Circle circle = (Circle)object;
```

11.9.3 Suppose `Fruit`, `Apple`, `Orange`, `GoldenDelicious`, and `McIntosh` are defined in the following inheritance hierarchy:



Assume the following code is given:

```
Fruit fruit = new GoldenDelicious();
Orange orange = new Orange();
```

Answer the following questions:

a. Is `fruit instanceof Fruit`?

b. Is `fruit instanceof Orange`?

c. Is `fruit instanceof Apple`?

d. Is `fruit instanceof GoldenDelicious`?

e. Is `fruit instanceof McIntosh`?

f. Is `orange instanceof Orange`?

g. Is `orange instanceof Fruit`?

h. Is `orange instanceof Apple`?

i. Suppose the method `makeAppleCider` is defined in the `Apple` class. Can `Fruit` invoke this method? Can `orange` invoke this method?

j. Suppose the method `makeOrangeJuice` is defined in the `Orange` class. Can `orange` invoke this method? Can `Fruit` invoke this method?

k. Is the statement `Orange p = new Apple()` legal?

l. Is the statement `McIntosh p = new Apple()` legal?

m. Is the statement `Apple p = new McIntosh()` legal?

11.9.4 What is wrong in the following code?

```java
public class Test {
  public static void main(String[] args) {
    Object fruit = new Fruit();
    Object apple = (Apple)fruit;
  }
}

class Apple extends Fruit {
}

class Fruit {
}
```

11.10 The Object's equals Method

Like the `toString()` method, the `equals(Object)` method is another useful method defined in the `Object` class.

Key Point

Another method defined in the `Object` class that is often used is the `equals` method. Its signature is

```java
public boolean equals(Object o)
```

This method tests whether two objects are equal. The syntax for invoking it is

```java
object1.equals(object2);
```

The default implementation of the `equals` method in the `Object` class is

```java
public boolean equals(Object obj) {
  return this == obj;
}
```

This implementation checks whether two reference variables point to the same object using the `==` operator. You should override this method in your custom class to test whether two distinct objects have the same content.

The `equals` method is overridden in many classes in the Java API, such as `java.lang.String` and `java.util.Date`, to compare whether the contents of two objects are equal. You have already used the `equals` method to compare two strings in Section 4.4.7, The `String` Class. The `equals` method in the `String` class is inherited from the `Object` class, and is overridden in the `String` class to test whether two strings are identical in content.

You can override the `equals` method in the `Circle` class to compare whether two circles are equal based on their radius as follows:

```
@Override
public boolean equals(Object o) {
  if (o instanceof Circle)
    return radius == ((Circle)o).radius;
  else
    return false;
}
```

== vs. equals

Note

The `==` comparison operator is used for comparing two primitive-data-type values or for determining whether two objects have the same references. The `equals` method is intended to test whether two objects have the same contents, provided the method is overridden in the defining class of the objects. The `==` operator is stronger than the `equals` method in that the `==` operator checks whether the two reference variables refer to the same object.

`equals(Object)`

Caution

Using the signature `equals(SomeClassName obj)` (e.g., `equals(Circle c)`) to override the `equals` method in a subclass is a common mistake. You should use `equals(Object obj)`. See CheckPoint Question 11.10.2.

Check Point

11.10.1 Does every object have a `toString` method and an `equals` method? Where do they come from? How are they used? Is it appropriate to override these methods?

11.10.2 When overriding the `equals` method, a common mistake is mistyping its signature in the subclass. For example, the `equals` method is incorrectly written as `equals(Circle circle)`, as shown in (a) in the following code; instead, it should be `equals(Object circle)`, as shown in (b). Show the output of running class `Test` with the `Circle` class in (a) and in (b), respectively.

```
public class Test {
  public static void main(String[] args) {
    Object circle1 = new Circle();
    Object circle2 = new Circle();
    System.out.println(circle1.equals(circle2));
  }
}
```

```
class Circle {
  double radius;

  public boolean equals(Circle circle) {
    return this.radius == circle.radius;
  }
}
```

(a)

```
class Circle {
  double radius;

  public boolean equals(Object o) {
    return this.radius ==
      ((Circle)o).radius;
  }
}
```

(b)

If `Object` is replaced by `Circle` in the `Test` class, what would be the output to run `Test` using the `Circle` class in (a) and (b), respectively?

11.11 The ArrayList Class

An `ArrayList` object can be used to store a list of objects.

Key Point

Now we are ready to introduce a very useful class for storing objects. You can create an array to store objects. However, once the array is created, its size is fixed. Java provides the `ArrayList` class, which can be used to store an unlimited number of objects. Figure 11.3 shows some methods in **`ArrayList`**.

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(e: E): void	Appends a new element `e` at the end of this list.
+add(index: int, e: E): void	Adds a new element `e` at the specified index in this list.
+clear(): void	Removes all elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int): E	Returns the element from this list at the specified index.
+indexOf(o: Object): int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object): int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the first element `CDT` from this list. Returns true if an element is removed.
+size(): int	Returns the number of elements in this list.
+remove(index: int): E	Removes the element at the specified index. Returns the removed element.
+set(index: int, e: E): E	Sets the element at the specified index.

FIGURE 11.3 An **`ArrayList`** stores an unlimited number of objects.

`ArrayList` is known as a generic class with a generic type **`E`**. You can specify a concrete type to replace **`E`** when creating an **`ArrayList`**. For example, the following statement creates an **`ArrayList`** and assigns its reference to variable **`cities`**. This **`ArrayList`** object can be used to store strings.

```
ArrayList<String> cities = new ArrayList<String>();
```

The following statement creates an **`ArrayList`** and assigns its reference to variable **`dates`**. This **`ArrayList`** object can be used to store dates.

```
ArrayList<java.util.Date> dates = new ArrayList<java.util.Date>();
```

Note

Since JDK 7, the statement

```
ArrayList <AConcreteType> list = new ArrayList<AConcreteType>();
```

can be simplified by

```
ArrayList<AConcreteType> list = new ArrayList<>();
```

type inference

The concrete type is no longer required in the constructor, thanks to a feature called *type inference*. The compiler is able to infer the type from the variable declaration. More discussions on generics including how to define custom generic classes and methods will be introduced in Chapter 19, Generics.

Listing 11.8 gives an example of using **`ArrayList`** to store objects.

LISTING 11.8 TestArrayList.java

import ArrayList
create ArrayList
add element
list size
contains element?
element index
is empty?
remove element
remove element
toString()
get element
create ArrayList

```java
import java.util.ArrayList;

public class TestArrayList {
  public static void main(String[] args) {
    // Create a list to store cities
    ArrayList<String> cityList = new ArrayList<>();

    // Add some cities in the list
    cityList.add("London");
    // cityList now contains [London]
    cityList.add("Denver");
    // cityList now contains [London, Denver]
    cityList.add("Paris");
    // cityList now contains [London, Denver, Paris]
    cityList.add("Miami");
    // cityList now contains [London, Denver, Paris, Miami]
    cityList.add("Seoul");
    // Contains [London, Denver, Paris, Miami, Seoul]
    cityList.add("Tokyo");
    // Contains [London, Denver, Paris, Miami, Seoul, Tokyo]

    System.out.println("List size? " + cityList.size());
    System.out.println("Is Miami in the list? " +
      cityList.contains("Miami"));
    System.out.println("The location of Denver in the list? "
      + cityList.indexOf("Denver"));
    System.out.println("Is the list empty? " +
      cityList.isEmpty()); // Print false

    // Insert a new city at index 2
    cityList.add(2, "Xian");
    // Contains [London, Denver, Xian, Paris, Miami, Seoul, Tokyo]

    // Remove a city from the list
    cityList.remove("Miami");
    // Contains [London, Denver, Xian, Paris, Seoul, Tokyo]

    // Remove a city at index 1
    cityList.remove(1);
    // Contains [London, Xian, Paris, Seoul, Tokyo]

    // Display the contents in the list
    System.out.println(cityList.toString());

    // Display the contents in the list in reverse order
    for (int i = cityList.size() - 1; i >= 0; i--)
      System.out.print(cityList.get(i) + " ");
    System.out.println();

    // Create a list to store two circles
    ArrayList<Circle> list = new ArrayList<>();

    // Add two circles
    list.add(new Circle(2));
    list.add(new Circle(3));

    // Display the area of the first circle in the list
    System.out.println("The area of the circle? " +
59          list.get(0).getArea());
60     }
61   }
```

```
List size? 6
Is Miami in the list? true
The location of Denver in the list? 1
Is the list empty? false
[London, Xian, Paris, Seoul, Tokyo]
Tokyo Seoul Paris Xian London
The area of the circle? 12.566370614359172
```

`add(Object)`

Since the `ArrayList` is in the `java.util` package, it is imported in line 1. The program creates an `ArrayList` of strings using its no-arg constructor and assigns the reference to `cityList` (line 6). The `add` method (lines 9–19) adds strings to the end of list. Thus, after `cityList.add("London")` (line 9), the list contains

```
[London]
```

After `cityList.add("Denver")` (line 11), the list contains

```
[London, Denver]
```

After adding `Paris`, `Miami`, `Seoul`, and `Tokyo` (lines 13–19), the list contains

```
[London, Denver, Paris, Miami, Seoul, Tokyo]
```

`size()`

Invoking `size()` (line 22) returns the size of the list, which is currently `6`. Invoking `contains("Miami")` (line 24) checks whether the object is in the list. In this case, it returns `true`, since `Miami` is in the list. Invoking `indexOf("Denver")` (line 26) returns the index of `Denver` in the list, which is `1`. If `Denver` were not in the list, it would return `-1`. The `isEmpty()` method (line 28) checks whether the list is empty. It returns `false`, since the list is not empty.

`add(index, Object)`

The statement `cityList.add(2, "Xian")` (line 31) inserts an object into the list at the specified index. After this statement, the list becomes

```
[London, Denver, Xian, Paris, Miami, Seoul, Tokyo]
```

`remove(Object)`

The statement `cityList.remove("Miami")` (line 35) removes the object from the list. After this statement, the list becomes

```
[London, Denver, Xian, Paris, Seoul, Tokyo]
```

`remove(index)`

The statement `cityList.remove(1)` (line 39) removes the object at the specified index from the list. After this statement, the list becomes

```
[London, Xian, Paris, Seoul, Tokyo]
```

The statement in line 43 is same as

```
System.out.println(cityList);
```

`toString()`

The `toString()` method returns a string representation of the list in the form of `[e0.toString(), e1.toString(), ..., ek.toString()]`, where `e0`, `e1`, . . . , and `ek` are the elements in the list.

`get(index)`

The `get(index)` method (line 47) returns the object at the specified index.

array vs. `ArrayList`

`ArrayList` objects can be used like arrays, but there are many differences. Table 11.1 lists their similarities and differences.

Once an array is created, its size is fixed. You can access an array element using the square-bracket notation (e.g., `a[index]`). When an `ArrayList` is created, its size is `0`.

TABLE 11.1 Differences and Similarities between Arrays and `ArrayList`

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

You cannot use the `get(index)` and `set(index, element)` methods if the element is not in the list. It is easy to add, insert, and remove elements in a list, but it is rather complex to add, insert, and remove elements in an array. You have to write code to manipulate the array in order to perform these operations. Note you can sort an array using the `java.util.Arrays.sort(array)` method. To sort an array list, use the `java.util.Collections.sort(arraylist)` method.

Suppose you want to create an `ArrayList` for storing integers. Can you use the following code to create a list?

```
ArrayList<int> listOfIntegers = new ArrayList<>();
```

No. This will not work because the elements stored in an `ArrayList` must be of an object type. You cannot use a primitive data type such as `int` to replace a generic type. However, you can create an `ArrayList` for storing `Integer` objects as follows:

```
ArrayList<Integer> listOfIntegers = new ArrayList<>();
```

remove(int) vs. remove(Integer)

Note the `remove(int index)` method removes an element at the specified index. To remove an integer value v from `listOfIntegers`, you need to use `listOfIntegers.remove(Integer.valueOf(v))`. This is not a good design in the Java API because it could easily lead to mistakes. It would be much better if `remove(int)` is renamed `removeAt(int)`.

Listing 11.9 gives a program that prompts the user to enter a sequence of numbers and displays the distinct numbers in the sequence. Assume the input ends with `0`, and `0` is not counted as a number in the sequence.

LISTING 11.9 `DistinctNumbers.java`

create an array list

```
import java.util.ArrayList;
import java.util.Scanner;

public class DistinctNumbers {
  public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<>();

    Scanner input = new Scanner(System.in);
    System.out.print("Enter integers (input ends with 0): ");
    int value;

    do {
      value = input.nextInt(); // Read a value from the input

15        if (!list.contains(value) && value != 0)
16          list.add(value); // Add the value if it is not in the list
17      } while (value != 0);
18
19      // Display the distinct numbers
20      System.out.print("The distinct integers are: ");
21      for (int i = 0; i < list.size(); i++)
22        System.out.print(list.get(i) + " ");
23    }
24  }
```

contained in list?

add to list

```
Enter numbers (input ends with 0): 1 2 3 2 1 6 3 4 5 4 5 1 2 3 0 ↵Enter
The distinct numbers are: 1 2 3 6 4 5
```

The program creates an **`ArrayList`** for **`Integer`** objects (line 6) and repeatedly reads a value in the loop (lines 12–17). For each value, if it is not in the list (line 15), add it to the list (line 16). You can rewrite this program using an array to store the elements rather than using an **`ArrayList`**. However, it is simpler to implement this program using an **`ArrayList`** for two reasons.

1. The size of an **`ArrayList`** is flexible so you don't have to specify its size in advance. When creating an array, its size must be specified.
2. **`ArrayList`** contains many useful methods. For example, you can test whether an element is in the list using the **`contains`** method. If you use an array, you have to write additional code to implement this method.

foreach loop

You can traverse the elements in an array using a foreach loop. The elements in an array list can also be traversed using a foreach loop using the following syntax:

```
for (elementType element: arrayList) {
  // Process the element
}
```

For example, you can replace the code in lines 20 and 21 using the following code:

```
for (Integer number: list)
  System.out.print(number + " ");
```

or

```
for (int number: list)
  System.out.print(number + " ");
```

Note the elements in **`list`** are **`Integer`** objects. They are automatically unboxed into **`int`** in this foreach loop.

Check Point

11.11.1 How do you do the following?

a. Create an **`ArrayList`** for storing double values?

b. Append an object to a list?

c. Insert an object at the beginning of a list?

d. Find the number of objects in a list?

e. Remove a given object from a list?

f. Remove the last object from a list?

g. Check whether a given object is in a list?

h. Retrieve an object at a specified index from a list?

11.11.2 Identify the errors in the following code.

```
ArrayList<String> list = new ArrayList<>();
list.add("Denver");
list.add("Austin");
list.add(new java.util.Date());
String city = list.get(0);
list.set(3, "Dallas");
System.out.println(list.get(3));
```

11.11.3 Suppose the `ArrayList list` contains `{"Dallas", "Dallas", "Houston", "Dallas"}`. What is the list after invoking `list.remove("Dallas")` one time? Does the following code correctly remove all elements with value `"Dallas"` from the list? If not, correct the code.

```
for (int i = 0; i < list.size(); i++)
  list.remove("Dallas");
```

11.11.4 Explain why the following code displays `[1, 3]` rather than `[2, 3]`.

```
ArrayList<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);
list.remove(1);
System.out.println(list);
```

How do you remove integer value `3` from the list?

11.11.5 Explain why the following code is wrong:

```
ArrayList<Double> list = new ArrayList<>();
list.add(1);
```

11.12 Useful Methods for Lists

Key Point

Java provides the methods for creating a list from an array, for sorting a list, and for finding maximum and minimum element in a list, and for shuffling a list.

array to array list

Often you need to create an array list from an array of objects or vice versa. You can write the code using a loop to accomplish this, but an easy way is to use the methods in the Java API. Here is an example to create an array list from an array:

```
String[] array = {"red", "green", "blue"};
ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
```

array list to array

The static method `asList` in the `Arrays` class returns a list that is passed to the `ArrayList` constructor for creating an `ArrayList`. Conversely, you can use the following code to create an array of objects from an array list:

```
String[] array1 = new String[list.size()];
list.toArray(array1);
```

sort a list

Invoking `list.toArray(array1)` copies the contents from `list` to `array1`. If the elements in a list are comparable, such as integers, double, or strings, you can use the static `sort` method in the `java.util.Collections` class to sort the elements. Here are some examples:

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
java.util.Collections.sort(list);
System.out.println(list);
```

You can use the static `max` and `min` in the `java.util.Collections` class to return the maximum and minimal element in a list. Here are some examples:

`max` and `min` methods

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
System.out.println(java.util.Collections.max(list));
System.out.println(java.util.Collections.min(list));
```

You can use the static `shuffle` method in the `java.util.Collections` class to perform a random shuffle for the elements in a list. Here are some examples:

`shuffle` method

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
java.util.Collections.shuffle(list);
System.out.println(list);
```

Check Point

11.12.1 Correct errors in the following statements:

```
int[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
```

11.12.2 Correct errors in the following statements:

```
int[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
System.out.println(java.util.Collections.max(array));
```

11.13 Case Study: A Custom Stack Class

This section designs a stack class for holding objects.

Key Point

Section 10.6 presented a stack class for storing `int` values. This section introduces a stack class to store objects. You can use an `ArrayList` to implement `Stack`, as shown in Listing 11.10. The UML diagram for the class is shown in Figure 11.4.

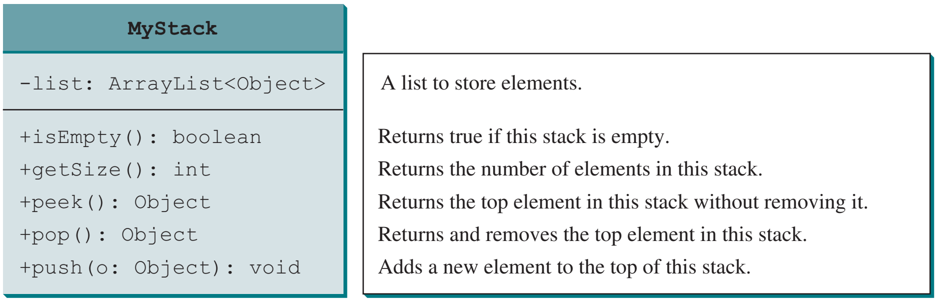

MyStack
-list: ArrayList<Object>
+isEmpty(): boolean
+getSize(): int
+peek(): Object
+pop(): Object
+push(o: Object): void
A list to store elements.
Returns true if this stack is empty.
Returns the number of elements in this stack.
Returns the top element in this stack without removing it.
Returns and removes the top element in this stack.
Adds a new element to the top of this stack.

FIGURE 11.4 The `MyStack` class encapsulates the stack storage and provides the operations for manipulating the stack.

LISTING 11.10 `MyStack.java`

```
import java.util.ArrayList;

public class MyStack {
  private ArrayList<Object> list = new ArrayList<>();

  public boolean isEmpty() {
    return list.isEmpty();
```

array list

stack empty?

get stack size

peek stack

remove

push

```
 8    }
 9
10    public int getSize() {
11      return list.size();
12    }
13
14    public Object peek() {
15      return list.get(getSize() - 1);
16    }
17
18    public Object pop() {
19      Object o = list.get(getSize() - 1);
20      list.remove(getSize() - 1);
21      return o;
22    }
23
24    public void push(Object o) {
25      list.add(o);
26    }
27
28    @Override
29    public String toString() {
30      return "stack: " + list.toString();
31    }
32  }
```

An array list is created to store the elements in the stack (line 4). The `isEmpty()` method (lines 6–8) returns `list.isEmpty()`. The `getSize()` method (lines 10–12) returns `list.size()`. The `peek()` method (lines 14–16) retrieves the element at the top of the stack without removing it. The end of the list is the top of the stack. The `pop()` method (lines 18–22) removes the top element from the stack and returns it. The `push(Object element)` method (lines 24–26) adds the specified element to the stack. The `toString()` method (lines 28–31) defined in the `Object` class is overridden to display the contents of the stack by invoking `list.toString()`. The `toString()` method implemented in `ArrayList` returns a string representation of all the elements in an array list.

composition

has-a

Design Guide

In Listing 11.10, `MyStack` contains `ArrayList`. The relationship between `MyStack` and `ArrayList` is *composition*. Composition essentially means declaring an instance variable for referencing an object. This object is said to be composed. While inheritance models an *is-a* relationship, composition models a *has-a* relationship. You could also implement `MyStack` as a subclass of `ArrayList` (see Programming Exercise 11.10). Using composition is better, however, because it enables you to define a completely new stack class without inheriting the unnecessary and inappropriate methods from `ArrayList`.

Check Point

11.13.1 Write statements that create a `MyStack` and add number `11` to the stack.

11.14 The protected Data and Methods

Key Point

A protected member of a class can be accessed from a subclass.

So far you have used the `private` and `public` keywords to specify whether data fields and methods can be accessed from outside of the class. Private members can be accessed only from inside of the class, and public members can be accessed from any other classes.

why `protected`?

Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not to allow nonsubclasses in different packages to access these data fields and methods. To accomplish this, you can use the `protected` keyword. This way you can access protected data fields or methods in a superclass from its subclasses.

The modifiers **`private`**, **`protected`**, and **`public`** are known as *visibility* or *accessibility modifiers* because they specify how classes and class members are accessed. The visibility of these modifiers increases in this order:

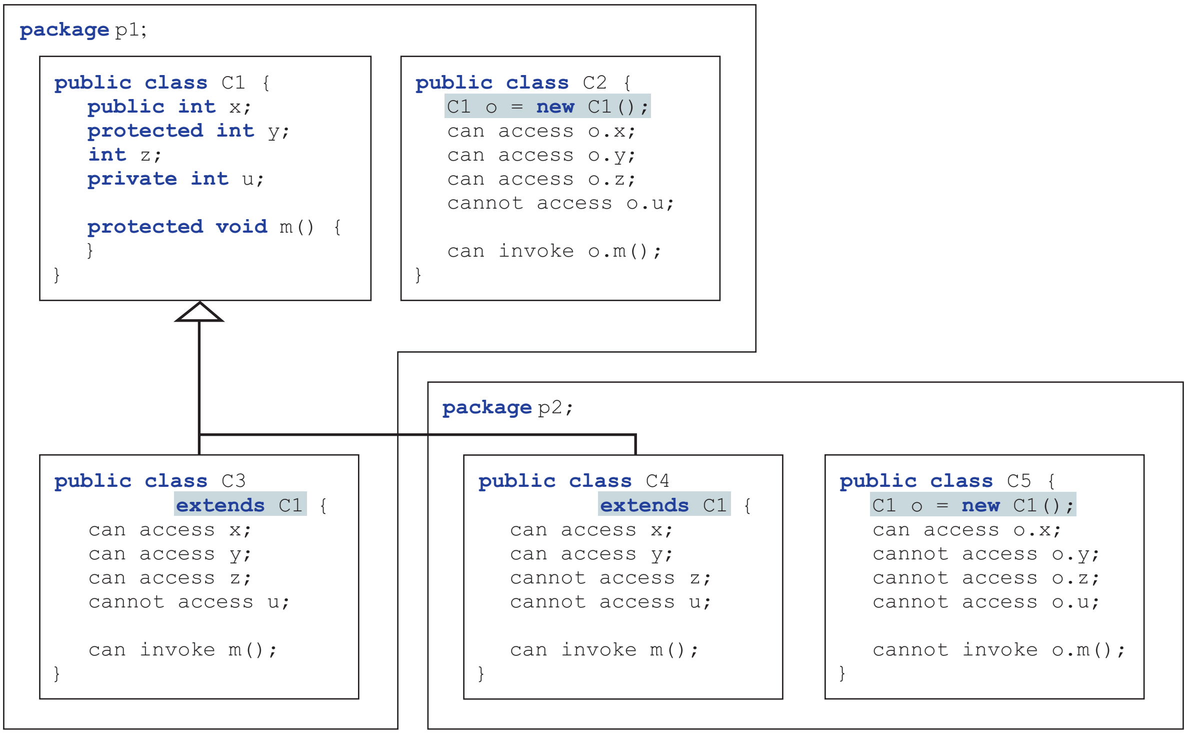
Visibility increases
⟶
private, default (no modifier), protected, public

Table 11.2 summarizes the accessibility of the members in a class. Figure 11.5 illustrates how a public, protected, default, and private datum or method in class **`C1`** can be accessed from a class **`C2`** in the same package, a subclass **`C3`** in the same package, a subclass **`C4`** in a different package, and a class **`C5`** in a different package.

Use the **`private`** modifier to hide the members of the class completely so they cannot be accessed directly from outside the class. Use no modifiers (the default) in order to allow the members of the class to be accessed directly from any class within the same package but not from other packages. Use the **`protected`** modifier to enable the members of the class to be accessed by the subclasses in any package or classes in the same package. Use the **`public`** modifier to enable the members of the class to be accessed by any class.

Table 11.2 Data and Methods Visibility

Modifier on Members in a Class	*Accessed from the Same Class*	*Accessed from the Same Package*	*Accessed from a Subclass in a Different Package*	*Accessed from a Different Package*
Public	✓	✓	✓	✓
Protected	✓	✓	✓	–
Default (no modifier)	✓	✓	–	–
Private	✓	–	–	–



```
package p1;

public class C1 {
   public int x;
   protected int y;
   int z;
   private int u;

   protected void m() {
   }
}
```

```
public class C2 {
   C1 o = new C1();
   can access o.x;
   can access o.y;
   can access o.z;
   cannot access o.u;

   can invoke o.m();
}
```

```
public class C3
         extends C1 {
   can access x;
   can access y;
   can access z;
   cannot access u;

   can invoke m();
}
```

```
package p2;

public class C4
         extends C1 {
   can access x;
   can access y;
   cannot access z;
   cannot access u;

   can invoke m();
}
```

```
public class C5 {
   C1 o = new C1();
   can access o.x;
   cannot access o.y;
   cannot access o.z;
   cannot access o.u;

   cannot invoke o.m();
}
```

Figure 11.5 Visibility modifiers are used to control how data and methods are accessed.

Your class can be used in two ways: (1) for creating instances of the class and (2) for defining subclasses by extending the class. Make the members `private` if they are not intended for use from outside the class. Make the members `public` if they are intended for the users of the class. Make the fields or methods `protected` if they are intended for the extenders of the class but not for the users of the class.

The `private` and `protected` modifiers can be used only for members of the class. The `public` modifier and the default modifier (i.e., no modifier) can be used on members of the class as well as on the class. A class with no modifier (i.e., not a public class) is not accessible by classes from other packages.

change visibility

Note

A subclass may override a protected method defined in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

Check Point

11.14.1 What modifier should you use on a class so a class in the same package can access it, but a class in a different package cannot access it?

11.14.2 What modifier should you use so a class in a different package cannot access the class, but its subclasses in any package can access it?

11.14.3 In the following code, the classes **A** and **B** are in the same package. If the question marks in (a) are replaced by blanks, can class **B** be compiled? If the question marks are replaced by `private`, can class **B** be compiled? If the question marks are replaced by `protected`, can class **B** be compiled?

```
package p1;

public class A {
  ___?___ int i;

  ___?___ void m() {
    ...
  }
}
```

(a)

```
package p1;

public class B extends A {
  public void m1(String[] args) {
    System.out.println(i);
    m();
  }
}
```

(b)

11.14.4 In the following code, the classes **A** and **B** are in different packages. If the question marks in (a) are replaced by blanks, can class **B** be compiled? If the question marks are replaced by `private`, can class **B** be compiled? If the question marks are replaced by `protected`, can class **B** be compiled?

```
package p1;

public class A {
  ___?___ int i;

  ___?___ void m() {
    ...
  }
}
```

(a)

```
package p2;

public class B extends A {
  public void m1(String[] args) {
    System.out.println(i);
    m();
  }
}
```

(b)

11.15 Preventing Extending and Overriding

Key Point

Neither a final class nor a final method can be extended. A final data field is a constant.

You may occasionally want to prevent classes from being extended. In such cases, use the **`final`** modifier to indicate a class is final and cannot be a parent class. The **`Math`** class is a final class. The **`String`**, **`StringBuilder`**, and **`StringBuffer`** classes, and all wrapper classes for primitive data types are also final classes. For example, the following class **`A`** is final and cannot be extended:

```
public final class A {
  // Data fields, constructors, and methods omitted
}
```

You also can define a method to be final; a final method cannot be overridden by its subclasses. For example, the following method **`m`** is final and cannot be overridden:

```
public class Test {
  // Data fields, constructors, and methods omitted

  public final void m() {
    // Do something
  }
}
```

> **Note**
>
> The modifiers **`public`**, **`protected`**, **`private`**, **`static`**, **`abstract`**, and **`final`** are used on classes and class members (data and methods), except that the **`final`** modifier can also be used on local variables in a method. A **`final`** local variable is a constant inside a method.

Check Point

11.15.1 How do you prevent a class from being extended? How do you prevent a method from being overridden?

11.15.2 Indicate true or false for the following statements:

a. A protected datum or method can be accessed by any class in the same package.

b. A protected datum or method can be accessed by any class in different packages.

c. A protected datum or method can be accessed by its subclasses in any package.

d. A final class can have instances.

e. A final class can be extended.

f. A final method can be overridden.

KEY TERMS

actual type	426
casting objects	429
constructor chaining	419
declared type	426
dynamic binding	426
inheritance	412
`instanceof`	430
is-a relationship	412
method overriding	421
multiple inheritance	418
override	421
polymorphism	425
`protected`	442
single inheritance	418
subclass	412
subtype	412
superclass	412
supertype	412
type inference	435

CHAPTER SUMMARY

1. You can define a new class from an existing class. This is known as class *inheritance*. The new class is called a *subclass*, *child class*, or *extended class*. The existing class is called a *superclass*, *parent class*, or *base class*.

2. A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited in the subclass. They can be invoked only from the constructors of the subclasses, using the keyword `super`.

3. A constructor may invoke an overloaded constructor or its superclass's constructor. The call must be the first statement in the constructor. If none of them is invoked explicitly, the compiler puts `super()` as the first statement in the constructor, which invokes the superclass's no-arg constructor.

4. To *override* a method, the method must be defined in the subclass using the same signature and the same or compatible return type as in its superclass.

5. An instance method can be overridden only if it is accessible. Thus, a private method cannot be overridden because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

6. Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

7. Every class in Java is descended from the `java.lang.Object` class. If no superclass is specified when a class is defined, its superclass is `Object`.

8. If a method's parameter type is a superclass (e.g., `Object`), you may pass an object to this method of any of the parameter's subclasses (e.g., `Circle` or `String`). This is known as polymorphism.

9. It is always possible to cast an instance of a subclass to a variable of a superclass because an instance of a subclass is *always* an instance of its superclass. When casting an instance of a superclass to a variable of its subclass, explicit casting must be used to confirm your intention to the compiler with the (`SubclassName`) cast notation.

10. A class defines a type. A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*.

11. When invoking an instance method from a reference variable, the *actual type of* the variable decides which implementation of the method is used *at runtime*. This is known as dynamic binding.

12. You can use `obj instanceof AClass` to test whether an object is an instance of a class.

13. You can use the `ArrayList` class to create an object to store a list of objects.

14. You can use the `protected` modifier to prevent the data and methods from being accessed by nonsubclasses from a different package.

15. You can use the `final` modifier to indicate a class is final and cannot be extended and to indicate a method is final and cannot be overridden.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

Sections 11.2–11.4

11.1 (*The `Triangle` class*) Design a class named `Triangle` that extends `GeometricObject`. The class contains:

- Three `double` data fields named `side1`, `side2`, and `side3` with default values `1.0` to denote three sides of a triangle.
- A no-arg constructor that creates a default triangle.
- A constructor that creates a triangle with the specified `side1`, `side2`, and `side3`.
- The accessor methods for all three data fields.
- A method named `getArea()` that returns the area of this triangle.
- A method named `getPerimeter()` that returns the perimeter of this triangle.
- A method named `toString()` that returns a string description for the triangle.

For the formula to compute the area of a triangle, see Programming Exercise 2.19. The `toString()` method is implemented as follows:

```
return "Triangle: side1 = " + side1 + " side2 = " + side2 +
  " side3 = " + side3;
```

Draw the UML diagrams for the classes `Triangle` and `GeometricObject` and implement the classes. Write a test program that prompts the user to enter three sides of the triangle, a color, and a Boolean value to indicate whether the triangle is filled. The program should create a `Triangle` object with these sides and set the `color` and `filled` properties using the input. The program should display the area, perimeter, color, and true or false to indicate whether it is filled or not.

Sections 11.5–11.14

11.2 (*The `Person`, `Student`, `Employee`, `Faculty`, and `Staff` classes*) Design a class named `Person` and its two subclasses named `Student` and `Employee`. Make `Faculty` and `Staff` subclasses of `Employee`. A person has a name, address, phone number, and e-mail address. A student has a class status (freshman, sophomore, junior, or senior). Define the status as a constant. An employee has an office, salary, and date hired. Use the `MyDate` class defined in Programming Exercise 10.14 to create an object for date hired. A faculty member has office hours and a rank. A staff member has a title. Override the `toString` method in each class to display the class name and the person's name.

Draw the UML diagram for the classes and implement them. Write a test program that creates a `Person`, `Student`, `Employee`, `Faculty`, and `Staff`, and invokes their `toString()` methods.

11.3 (*Subclasses of `Account`*) In Programming Exercise 9.7, the `Account` class was defined to model a bank account. An account has the properties account number, balance, annual interest rate, and date created, and methods to deposit and withdraw funds. Create two subclasses for checking and saving accounts. A checking account has an overdraft limit, but a savings account cannot be overdrawn.

Draw the UML diagram for the classes and implement them. Write a test program that creates objects of `Account`, `SavingsAccount`, and `CheckingAccount` and invokes their `toString()` methods.

11.4 (*Maximum element in* `ArrayList`) Write the following method that returns the maximum value in an `ArrayList` of integers. The method returns `null` if the list is `null` or the list size is `0`.

```
public static Integer max(ArrayList<Integer> list)
```

Write a test program that prompts the user to enter a sequence of numbers ending with `0` and invokes this method to return the largest number in the input.

11.5 (*The* `Course` *class*) Rewrite the `Course` class in Listing 10.6. Use an `ArrayList` to replace an array to store students. Draw the new UML diagram for the class. You should not change the original contract of the `Course` class (i.e., the definition of the constructors and methods should not be changed, but the private members may be changed.)

11.6 (*Use* `ArrayList`) Write a program that creates an `ArrayList` and adds a `Loan` object, a `Date` object, a string, and a `Circle` object to the list, and use a loop to display all the elements in the list by invoking the object's `toString()` method.

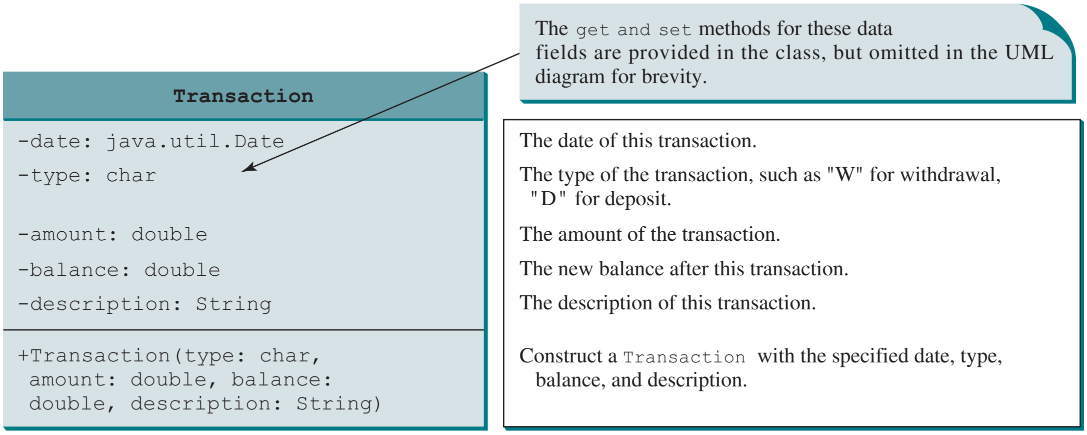
11.7 (*Shuffle* `ArrayList`) Write the following method that shuffles the elements in an `ArrayList` of integers:

```
public static void shuffle(ArrayList<Integer> list)
```

VideoNote
New Account class

****11.8** (*New* `Account` *class*) An `Account` class was specified in Programming Exercise 9.7. Design a new `Account` class as follows:

- Add a new data field `name` of the `String` type to store the name of the customer.
- Add a new constructor that constructs an account with the specified name, id, and balance.
- Add a new data field named `transactions` whose type is `ArrayList` that stores the transaction for the accounts. Each transaction is an instance of the `Transaction` class, which is defined as shown in Figure 11.6.



The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The date of this transaction.

The type of the transaction, such as "W" for withdrawal, "D" for deposit.

The amount of the transaction.

The new balance after this transaction.

The description of this transaction.

Construct a Transaction with the specified date, type, balance, and description.

FIGURE 11.6 The `Transaction` class describes a transaction for a bank account.

- Modify the `withdraw` and `deposit` methods to add a transaction to the `transactions` array list.
- All other properties and methods are the same as in Programming Exercise 9.7.

Write a test program that creates an `Account` with annual interest rate `1.5%`, balance `1000`, id `1122`, and name `George`. Deposit $30, $40, and $50 to the account and withdraw $5, $4, and $2 from the account. Print an account summary that shows the account holder name, interest rate, balance, and all transactions.

***11.9** (*Largest rows and columns*) Write a program that randomly fills in `0`s and `1`s into an n-by-n matrix, prints the matrix, and finds the rows and columns with the most `1`s. (*Hint*: Use two `ArrayList`s to store the row and column indices with the most `1`s.) Here is a sample run of the program:

```
Enter the array size n: 4 ↵Enter
The random array is
0011
0011
1101
1010
The largest row index: 2
The largest column index: 2, 3
```

11.10 (*Implement* `MyStack` *using inheritance*) In Listing 11.10, `MyStack` is implemented using composition. Define a new stack class that extends `ArrayList`.

Draw the UML diagram for the classes then implement `MyStack`. Write a test program that prompts the user to enter five strings and displays them in reverse order.

11.11 (*Sort* `ArrayList`) Write the following method that sorts an `ArrayList` of numbers:

```
public static void sort(ArrayList<Integer> list)
```

Write a test program that prompts the user to enter five numbers, stores them in an array list, and displays them in increasing order.

11.12 (*Sum* `ArrayList`) Write the following method that returns the sum of all numbers in an `ArrayList`:

```
public static double sum(ArrayList<Double> list)
```

Write a test program that prompts the user to enter five numbers, stores them in an array list, and displays their sum.

***11.13** (*Remove duplicates*) Write a method that removes the duplicate elements from an array list of integers using the following header:

```
public static void removeDuplicate(ArrayList<Integer> list)
```

Write a test program that prompts the user to enter 10 integers to a list and displays the distinct integers in their input order and separated by exactly one space. Here is a sample run:

```
Enter 10 integers: 34 5 3 5 6 4 33 2 2 4 ↵Enter
The distinct integers are 34 5 3 6 4 33 2
```

11.14 (*Combine two lists*) Write a method that returns the union of two array lists of integers using the following header:

```
public static ArrayList<Integer> union(
  ArrayList<Integer> list1, ArrayList<Integer> list2)
```

For example, the addition of two array lists {2, 3, 1, 5} and {3, 4, 6} is {2, 3, 1, 5, 3, 4, 6}. Write a test program that prompts the user to enter two lists, each with five integers, and displays their union. The numbers are separated by exactly one space. Here is a sample run:

```
Enter five integers for list1: 3 5 45 4 3 ↵Enter
Enter five integers for list2: 33 51 5 4 13 ↵Enter
The combined list is 3 5 45 4 3 33 51 5 4 13
```

***11.15** (*Area of a convex polygon*) A polygon is convex if it contains any line segments that connects two points of the polygon. Write a program that prompts the user to enter the number of points in a convex polygon, enter the points clockwise, then displays the area of the polygon. For the formula for computing the area of a polygon, see http://www.mathwords.com/a/area_convex_polygon.htm. Here is a sample run of the program:

```
Enter the number of points: 7 ↵Enter
Enter the coordinates of the points:
  -12 0 -8.5 10 0 11.4 5.5 7.8 6 -5.5 0 -7 -3.5 -5.5 ↵Enter
The total area is 244.57
```

****11.16** (*Addition quiz*) Rewrite Listing 5.1, RepeatAdditionQuiz.java, to alert the user if an answer is entered again. (*Hint:* use an array list to store answers.) Here is a sample run of the program:

```
What is 5 + 9? 12 ↵Enter
Wrong answer. Try again. What is 5 + 9? 34 ↵Enter
Wrong answer. Try again. What is 5 + 9? 12 ↵Enter
You already entered 12
Wrong answer. Try again. What is 5 + 9? 14 ↵Enter
You got it!
```

****11.17** (*Algebra: perfect square*) Write a program that prompts the user to enter an integer `m` and find the smallest integer `n` such that `m * n` is a perfect square. (*Hint:* Store all smallest factors of `m` into an array list. `n` is the product of the factors that appear an odd number of times in the array list. For example, consider `m` = 90, store the factors 2, 3, 3, and 5 in an array list. 2 and 5 appear an odd number of times in the array list. Thus, `n` is 10.) Here is a sample run of the program:

```
Enter an integer m: 1500 ↵Enter
The smallest number n for m * n to be a perfect square is 15
m * n is 22500
```

```
Enter an integer m: 63 ↵Enter
The smallest number n for m * n to be a perfect square is 7
m * n is 441
```

****11.18** (*`ArrayList` of Character*) Write a method that returns an array list of `Character` from a string using the following header:

```
public static ArrayList<Character> toCharacterArray(String s)
```

For example, `toCharacterArray("abc")` returns an array list that contains characters `'a'`, `'b'`, and `'c'`.

****11.19** (*Bin packing using first fit*) The bin packing problem is to pack the objects of various weights into containers. Assume each container can hold a maximum of 10 pounds. The program uses an algorithm that places an object into the first bin in which it would fit. Your program should prompt the user to enter the total number of objects and the weight of each object. The program displays the total number of containers needed to pack the objects and the contents of each container. Here is a sample run of the program:

```
Enter the number of objects: 6
Enter the weights of the objects: 7 5 2 3 5 8
Container 1 contains objects with weight 7 2
Container 2 contains objects with weight 5 3
Container 3 contains objects with weight 5
Container 4 contains objects with weight 8
```

Does this program produce an optimal solution, that is, finding the minimum number of containers to pack the objects?

CHAPTER

12

EXCEPTION HANDLING AND TEXT I/O

Objectives

- To get an overview of exceptions and exception handling (§12.2).
- To explore the advantages of using exception handling (§12.2).
- To distinguish exception types: `Error` (fatal) vs. `Exception` (nonfatal) and checked vs. unchecked (§12.3).
- To declare exceptions in a method header (§12.4.1).
- To throw exceptions in a method (§12.4.2).
- To write a `try-catch` block to handle exceptions (§12.4.3).
- To explain how an exception is propagated (§12.4.3).
- To obtain information from an exception object (§12.4.4).
- To develop applications with exception handling (§12.4.5).
- To use the `finally` clause in a `try-catch` block (§12.5).
- To use exceptions only for unexpected errors (§12.6).
- To rethrow exceptions in a `catch` block (§12.7).
- To create chained exceptions (§12.8).
- To define custom exception classes (§12.9).
- To discover file/directory properties, to delete and rename files/directories, and to create directories using the `File` class (§12.10).
- To write data to a file using the `PrintWriter` class (§12.11.1).
- To use try-with-resources to ensure that the resources are closed automatically (§12.11.2).
- To read data from a file using the `Scanner` class (§12.11.3).
- To understand how data is read using a `Scanner` (§12.11.4).
- To develop a program that replaces text in a file (§12.11.5).
- To read data from the Web (§12.12).
- To develop a Web crawler (§12.13).

12.1 Introduction

Key Point

Exceptions are runtime errors. Exception handling enables a program to deal with runtime errors and continue its normal execution.

Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out. For example, if you access an array using an index that is out of bounds, you will get a runtime error with an `ArrayIndexOutOfBoundsException`. If you enter a `double` value when your program expects an integer, you will get a runtime error with an `InputMismatchException`.

exception

In Java, runtime errors are thrown as exceptions. An *exception* is an object that represents an error or a condition that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally. How can you handle the exception so the program can continue to run or else terminate gracefully? This chapter introduces this subject, and text input and output.

12.2 Exception-Handling Overview

Key Point

VideoNote

Exception-handling advantages

Exceptions are thrown from a method. The caller of the method can catch and handle the exception.

To demonstrate exception handling, including how an exception object is created and thrown, let's begin with the example in Listing 12.1, which reads in two integers and displays their quotient.

Listing 12.1 Quotient.java

read two integers

integer division

```java
import java.util.Scanner;

public class Quotient {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    System.out.println(number1 + " / " + number2 + " is " +
      (number1 / number2));
  }
}
```

```
Enter two integers: 5 2 ↵Enter
5 / 2 is 2
```

```
Enter two integers: 3 0 ↵Enter
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Quotient.main(Quotient.java:13)
```

If you entered `0` for the second number, a runtime error would occur, because you cannot divide an integer by `0`. (*Note a floating-point number divided by* `0` *does not raise an exception.*) A simple way to fix this error is to add an `if` statement to test the second number, as shown in Listing 12.2.

Listing 12.2 QuotientWithIf.java

```
import java.util.Scanner;

public class QuotientWithIf {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    if (number2 != 0)
      System.out.println(number1 + " / " + number2
        + " is " + (number1 / number2));
    else
      System.out.println("Divisor cannot be zero ");
  }
}
```

read two integers

test `number2`

```
Enter two integers: 5 0 ↵Enter
Divisor cannot be zero
```

Before introducing exception handling, let us rewrite Listing 12.2 to compute a quotient using a method, as shown in Listing 12.3.

Listing 12.3 QuotientWithMethod.java

```
import java.util.Scanner;

public class QuotientWithMethod {
  public static int quotient(int number1, int number2) {
    if (number2 == 0) {
      System.out.println("Divisor cannot be zero");
      System.exit(1);
    }

    return number1 / number2;
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    int result = quotient(number1, number2);
    System.out.println(number1 + " / " + number2 + " is "
      + result);
  }
}
```

quotient method

terminate the program

read two integers

invoke method

```
Enter two integers: 5 3 ↵Enter
5 / 3 is 1
```

```
Enter two integers: 5 0 ↵Enter
Divisor cannot be zero
```

The method **`quotient`** (lines 4–11) returns the quotient of two integers. If **`number2`** is **`0`**, it cannot return a value, so the program is terminated in line 7. This is clearly a problem. You should not let the method terminate the program—the *caller* should decide whether to terminate the program.

How can a method notify its caller when an exception has occurred? Java enables a method to throw an exception that can be caught and handled by the caller. Listing 12.3 can be rewritten, as shown in Listing 12.4.

LISTING 12.4 QuotientWithException.java

```java
import java.util.Scanner;

public class QuotientWithException {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    return number1 / number2;
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    try {
      int result = quotient(number1, number2);
      System.out.println(number1 + " / " + number2 + " is "
        + result);
    }
    catch (ArithmeticException ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }

    System.out.println("Execution continues ...");
  }
}
```

If an `ArithmeticException` occurs

quotient method

throw exception

read two integers

`try` block

invoke method

`catch` block

```
Enter two integers: 5 3 ↵Enter
5 / 3 is 1
Execution continues ...
```

```
Enter two integers: 5 0 ↵Enter
Exception: an integer cannot be divided by zero
Execution continues ...
```

If `number2` is `0`, the method throws an exception (line 6) by executing

```
throw new ArithmeticException("Divisor cannot be zero");
```

throw statement

The value thrown, in this case `new ArithmeticException("Divisor cannot be zero")`, is called an *exception*. The execution of a `throw` statement is called *throwing an exception*. The exception is an object created from an exception class. In this case, the exception class is `java.lang.ArithmeticException`. The constructor `ArithmeticException(str)` is invoked to construct an exception object, where `str` is a message that describes the exception.

exception

throw exception

When an exception is thrown, the normal execution flow is interrupted. As the name suggests, to "throw an exception" is to pass the exception from one place to another. The statement for invoking the method is contained in a `try` block. The `try` block (lines 19–23) contains the code that is executed in normal circumstances. The exception is caught by the `catch` block. The code in the `catch` block is executed to *handle the exception*. Afterward, the statement (line 29) after the `catch` block is executed.

handle exception

The `throw` statement is analogous to a method call, but instead of calling a method, it calls a `catch` block. In this sense, a `catch` block is like a method definition with a parameter that matches the type of the value being thrown. Unlike a method, however, after the `catch` block is executed, the program control does not return to the `throw` statement; instead, it executes the next statement after the `catch` block.

The identifier `ex` in the `catch`–block header

```
catch (ArithmeticException ex)
```

acts very much like a parameter in a method. Thus, this parameter is referred to as a `catch`–block parameter. The type (e.g., `ArithmeticException`) preceding `ex` specifies what kind of exception the `catch` block can catch. Once the exception is caught, you can access the thrown value from this parameter in the body of a `catch` block.

`catch`–block parameter

In summary, a template for a `try`-`throw`-`catch` block may look as follows:

```
try {
  Code to run;
  A statement or a method that may throw an exception;
  More code to run;
}
catch (type ex) {
  Code to process the exception;
}
```

An exception may be thrown directly by using a `throw` statement in a `try` block, or by invoking a method that may throw an exception.

The main method invokes `quotient` (line 20). If the quotient method executes normally, it returns a value to the caller. If the `quotient` method encounters an exception, it throws the exception back to its caller. The caller's `catch` block handles the exception.

Now you can see the *advantage* of using exception handling: It enables a method to throw an exception to its caller, enabling the caller to handle the exception. Without this capability, the called method itself must handle the exception or terminate the program. Often the called method does not know what to do in case of error. This is typically the case for the library methods. The library method can detect the error, but only the caller

advantage

knows what needs to be done when an error occurs. The key benefit of exception handling is separating the detection of an error (done in a called method) from the handling of an error (done in the calling method).

Many library methods throw exceptions. Listing 12.5 gives an example that handles an `InputMismatchException` when reading an input.

LISTING 12.5 `InputMismatchExceptionDemo.java`

create a Scanner

try block

catch block

If an `InputMismatch` `Exception` occurs

```
1  import java.util.*;
2
3  public class InputMismatchExceptionDemo {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      boolean continueInput = true;
7
8      do {
9        try {
10         System.out.print("Enter an integer: ");
11         int number = input.nextInt();
12
13         // Display the result
14         System.out.println(
15           "The number entered is " + number);
16
17         continueInput = false;
18       }
19       catch (InputMismatchException ex) {
20         System.out.println("Try again. (" +
21           "Incorrect input: an integer is required)");
22         input.nextLine(); // Discard input
23       }
24     } while (continueInput);
25   }
26 }
```

```
Enter an integer: 3.5 ↵Enter
Try again. (Incorrect input: an integer is required)
Enter an integer: 4 ↵Enter
The number entered is 4
```

When executing `input.nextInt()` (line 11), an `InputMismatchException` occurs if the input entered is not an integer. Suppose `3.5` is entered. An `InputMismatchException` occurs and the control is transferred to the `catch` block. The statements in the `catch` block are now executed. The statement `input.nextLine()` in line 22 discards the current input line so the user can enter a new line of input. The variable `continueInput` controls the loop. Its initial value is `true` (line 6) and it is changed to `false` (line 17) when a valid input is received. Once a valid input is received, there is no need to continue the input.

Check Point

12.2.1 What is the advantage of using exception handling?

12.2.2 Which of the following statements will throw an exception?

```
System.out.println(1 / 0);
System.out.println(1.0 / 0);
```

12.2.3 Point out the problem in the following code. Does the code throw any exceptions?

```
long value = Long.MAX_VALUE + 1;
System.out.println(value);
```

12.2.4 What does the JVM do when an exception occurs? How do you catch an exception?

12.2.5 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    try {
      int value = 30;
      if (value < 40)
        throw new Exception("value is too small");
    }
    catch (Exception ex) {
      System.out.println(ex.getMessage());
    }
    System.out.println("Continue after the catch block");
  }
}
```

What would be the output if the line

`int value = 30;`

were changed to

`int value = 50;`

12.2.6 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    for (int i = 0; i < 2; i++) {
      System.out.print(i + " ");
      try {
        System.out.println(1 / 0);
      }
      catch (Exception ex) {
      }
    }
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    try {
      for (int i = 0; i < 2; i++) {
        System.out.print(i + " ");
        System.out.println(1 / 0);
      }
    }
    catch (Exception ex) {
    }
  }
}
```

(b)

12.3 Exception Types

Key Point

Exceptions are objects, and objects are defined using classes. The root class for exceptions is `java.lang.Throwable`.

The preceding section used the classes `ArithmeticException` and `InputMismatchException`. Are there any other types of exceptions you can use? Can you define your own exception classes? Yes. There are many predefined exception classes in the Java API. Figure 12.1 shows some of them, and in Section 12.9, you will learn how to define your own exception classes.

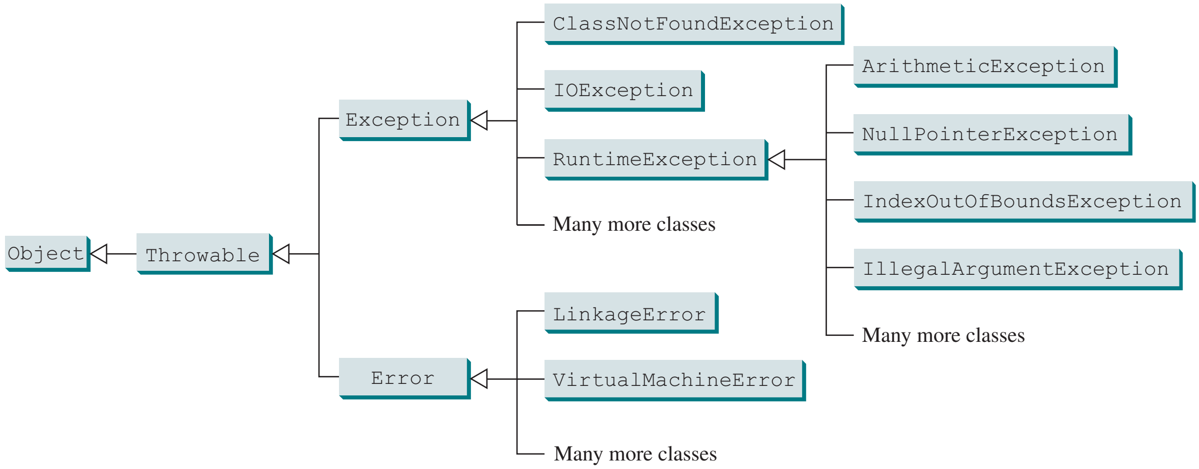

Object
Throwable
Exception
Error
ClassNotFoundException
IOException
RuntimeException
Many more classes
ArithmeticException
NullPointerException
IndexOutOfBoundsException
IllegalArgumentException
Many more classes
LinkageError
VirtualMachineError
Many more classes

Figure 12.1 Exceptions thrown are instances of the classes shown in this diagram, or of subclasses of one of these classes.

Note

The class names `Error`, `Exception`, and `RuntimeException` are somewhat confusing. All three of these classes are exceptions and all of the errors occur at runtime.

The `Throwable` class is the root of exception classes. All Java exception classes inherit directly or indirectly from `Throwable`. You can create your own exception classes by extending `Exception` or a subclass of `Exception`.

The exception classes can be classified into three major types: system errors, exceptions, and runtime exceptions.

system error

- *System errors* are thrown by the JVM and are represented in the `Error` class. The `Error` class describes internal system errors, though such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully. Examples of subclasses of `Error` are listed in Table 12.1.

Table 12.1 Examples of Subclasses of `Error`

Class	*Reasons for Exception*
`LinkageError`	A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
`VirtualMachineError`	The JVM is broken or has run out of the resources it needs in order to continue operating.

exception

- *Exceptions* are represented in the `Exception` class, which describes errors caused by your program and by external circumstances. These errors can be caught and handled by your program. Examples of subclasses of `Exception` are listed in Table 12.2.

Table 12.2 Examples of Subclasses of `Exception`

Class	*Reasons for Exception*
`ClassNotFoundException`	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the `java` command or if your program were composed of, say, three class files, only two of which could be found.
`IOException`	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of `IOException` are `InterruptedIOException`, `EOFException` (EOF is short for End of File), and `FileNotFoundException`.

- *Runtime exceptions* are represented in the `RuntimeException` class, which describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors. Runtime exceptions normally indicate programming errors. Examples of subclasses are listed in Table 12.3.

runtime exception

Table 12.3 Examples of Subclasses of `RuntimeException`

Class	*Reasons for Exception*
`ArithmeticException`	Dividing an integer by zero. Note floating-point arithmetic does not throw exceptions (see Appendix E, Special Floating-Point Values).
`NullPointerException`	Attempt to access an object through a `null` reference variable.
`IndexOutOfBoundsException`	Index to an array is out of range.
`IllegalArgumentException`	A method has passed an argument that is illegal or inappropriate.

`RuntimeException`, `Error`, and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning the compiler forces the programmer to check and deal with them in a `try-catch` block or declare it in the method header. Declaring an exception in the method header will be covered in Section 12.4.

unchecked exception
checked exception

In most cases, unchecked exceptions reflect programming logic errors that are unrecoverable. For example, a `NullPointerException` is thrown if you access an object through a reference variable before an object is assigned to it; an `IndexOutOfBoundsException` is thrown if you access an element in an array outside the bounds of the array. These are logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in a program. To avoid cumbersome overuse of `try-catch` blocks, Java does not mandate that you write code to catch or declare unchecked exceptions.

Check Point

12.3.1 Describe the Java `Throwable` class, its subclasses, and the types of exceptions.

12.3.2 What `RuntimeException` will the following programs throw, if any?

```
public class Test {
  public static void main(String[] args) {
    System.out.println(1 / 0);
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    int[] list = new int[5];
    System.out.println(list[5]);
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
    String s = "abc";
    System.out.println(s.charAt(3));
  }
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    Object o = new Object();
    String d = (String)o;
  }
}
```

(d)

```
public class Test {
  public static void main(String[] args) {
    Object o = null;
    System.out.println(o.toString());
  }
}
```

(e)

```
public class Test {
  public static void main(String[] args) {
    System.out.println(1.0 / 0);
  }
}
```

(f)

12.3.3 What is a checked exception and what is an unchecked exception?

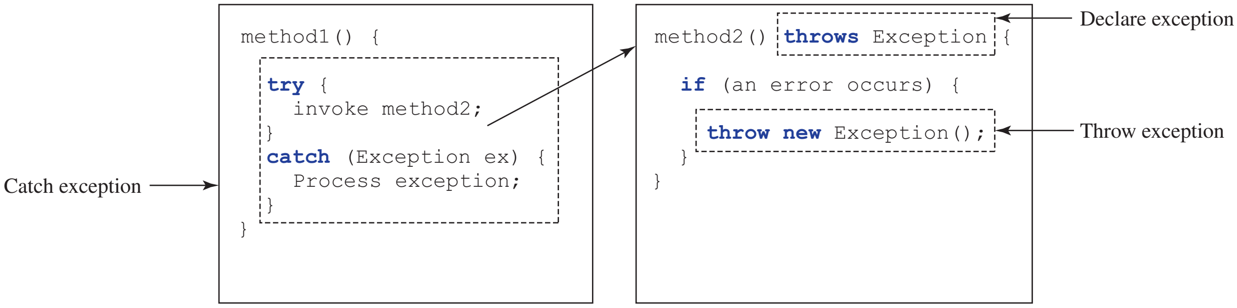
12.4 Declaring, Throwing, and Catching Exceptions

Key Point

A handler for an exception is found by propagating the exception backward through a chain of method calls, starting from the current method.

The preceding sections gave you an overview of exception handling and introduced several predefined exception types. This section provides an in-depth discussion of exception handling.

Java's exception-handling model is based on three operations: *declaring an exception*, *throwing an exception*, and *catching an exception*, as shown in Figure 12.2.



```
method1() {

  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

Figure 12.2 Exception handling in Java consists of declaring exceptions, throwing exceptions, and catching and processing exceptions.

12.4.1 Declaring Exceptions

declare exception

In Java, the statement currently being executed belongs to a method. The Java interpreter invokes the `main` method to start executing a program. Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*. Because system errors and runtime errors can happen to any code, Java does not require that you declare `Error` and `RuntimeException` (unchecked exceptions) explicitly in the method. However, all other exceptions thrown by the method must be explicitly declared in the method header so the caller of the method is informed of the exception.

To declare an exception in a method, use the `throws` keyword in the method header, as in this example:

```
public void myMethod() throws IOException
```

The `throws` keyword indicates `myMethod` might throw an `IOException`. If the method might throw multiple exceptions, add a list of the exceptions, separated by commas, after `throws`:

```
public void myMethod()
  throws Exception1, Exception2, ..., ExceptionN
```

Note

If a method does not declare exceptions in the superclass, you cannot override it to declare exceptions in the subclass.

12.4.2 Throwing Exceptions

throw exception

A program that detects an error can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example: Suppose the program detects that an argument passed to the method violates the method contract (e.g., the argument must be nonnegative, but a negative argument is passed); the program can create an instance of `IllegalArgumentException` and throw it, as follows:

```
IllegalArgumentException ex =
  new IllegalArgumentException("Wrong Argument");
throw ex;
```

Or, if you prefer, you can use the following:

```
throw new IllegalArgumentException("Wrong Argument");
```

Note

`IllegalArgumentException` is an exception class in the Java API. In general, each exception class in the Java API has at least two constructors: a no-arg constructor and a constructor with a `String` argument that describes the exception. This argument is called the *exception message*, which can be obtained by invoking `getMessage()` from an exception object.

exception message

Tip

The keyword to declare an exception is `throws`, and the keyword to throw an exception is `throw`.

`throws` vs. `throw`

12.4.3 Catching Exceptions

You now know how to declare an exception and how to throw an exception. When an exception is thrown, it can be caught and handled in a `try-catch` block, as follows:

catch exception

```
try {
  statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVarN) {
  handler for exceptionN;
}
```

If no exceptions arise during the execution of the `try` block, the `catch` blocks are skipped.

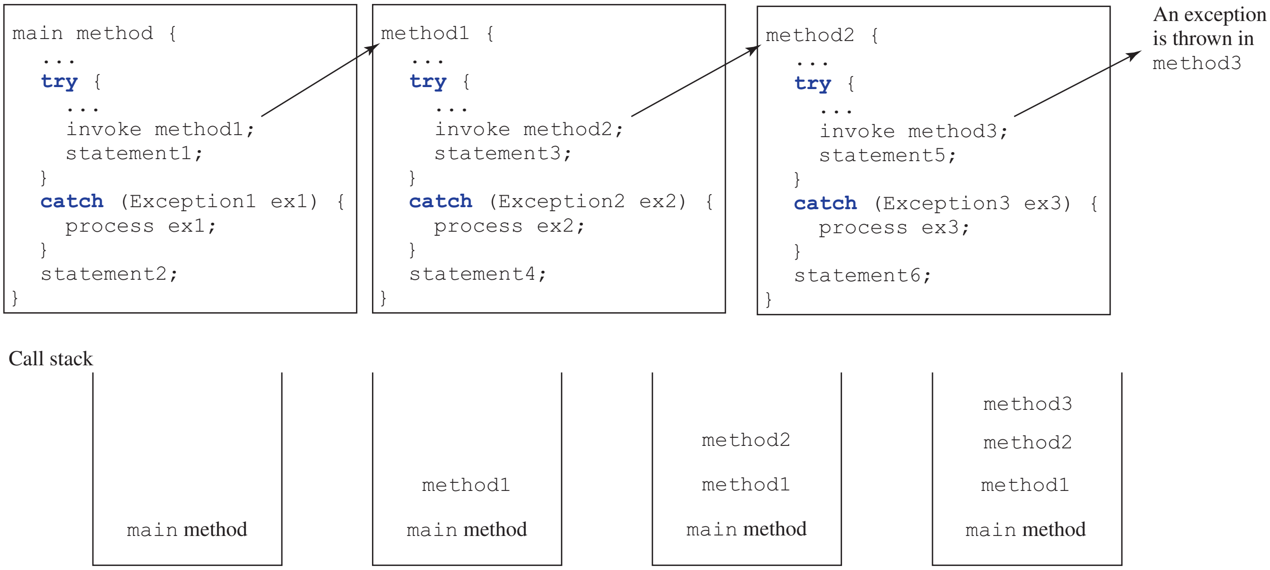
If one of the statements inside the `try` block throws an exception, Java skips the remaining statements in the `try` block and starts the process of finding the code to handle the exception. The code that handles the exception is called the *exception handler*; it is found by *propagating the exception* backward through a chain of method calls, starting from the current method. Each `catch` block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the `catch` block. If so, the exception object is assigned to the variable declared and the code in the `catch` block is executed. If no handler is found, Java exits this method, passes the exception to the method's caller, and continues the same process to find a handler. If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console. The process of finding a handler is called *catching an exception*.

exception handler

exception propagation

Suppose the `main` method invokes `method1`, `method1` invokes `method2`, `method2` invokes `method3`, and `method3` throws an exception, as shown in Figure 12.3. Consider the following scenario:

- If the exception type is `Exception3`, it is caught by the `catch` block for handling exception `ex3` in `method2`. `statement5` is skipped and `statement6` is executed.
- If the exception type is `Exception2`, `method2` is aborted, the control is returned to `method1`, and the exception is caught by the `catch` block for handling exception `ex2` in `method1`. `statement3` is skipped and `statement4` is executed.
- If the exception type is `Exception1`, `method1` is aborted, the control is returned to the `main` method, and the exception is caught by the `catch` block for handling exception `ex1` in the `main` method. `statement1` is skipped and `statement2` is executed.
- If the exception type is not caught in `method2`, `method1`, or `main`, the program terminates and `statement1` and `statement2` are not executed.



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    process ex3;
  }
  statement6;
}
```

FIGURE 12.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

catch block

Note

Various exception classes can be derived from a common superclass. If a `catch` block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass.

order of exception handlers

Note

The order in which exceptions are specified in `catch` blocks is important. A compile error will result if a catch block for a superclass type appears before a catch block for a subclass type. For example, the ordering in (a) below is erroneous, because `RuntimeException` is a subclass of `Exception`. The correct ordering should be as shown in (b).

```
try {
  ...
}
catch (Exception ex) {
  ...
}
catch (RuntimeException ex) {
  ...
}
```

(a) Wrong order

```
try {
  ...
}
catch (RuntimeException ex) {
  ...
}
catch (Exception ex) {
  ...
}
```

(b) Correct order

Note

catch or declare checked exceptions

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than `Error` or `RuntimeException`), you must invoke it in a `try-catch` block or declare to throw the exception in the calling method. For example, suppose method `p1` invokes method `p2` and `p2` may throw a checked exception (e.g., `IOException`); you have to write the code as shown in (a) or (b) below.

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a) Catch exception

```
void p1() throws IOException {

  p2();

}
```

(b) Throw exception

Note

JDK 7 multicatch

You can use the new JDK 7 multicatch feature to simplify coding for the exceptions with the same handling code. The syntax is:

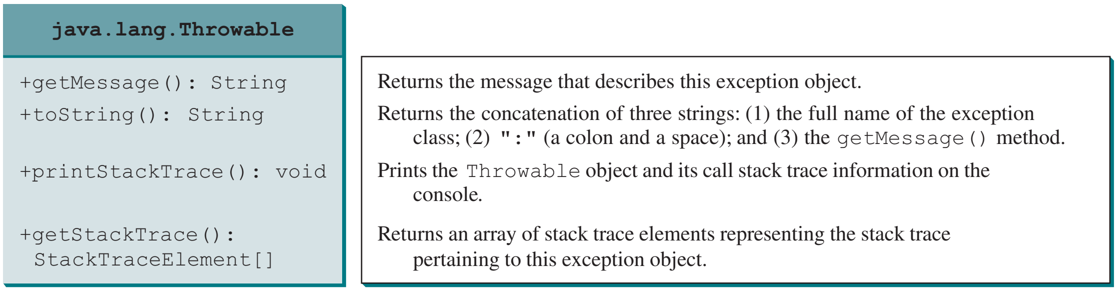
```
catch (Exception1 | Exception2 | ... | Exceptionk ex) {
  // Same code for handling these exceptions
}
```

Each exception type is separated from the next with a vertical bar (`|`). If one of the exceptions is caught, the handling code is executed.

12.4.4 Getting Information from Exceptions

methods in Throwable

An exception object contains valuable information about the exception. You may use the following instance methods in the `java.lang.Throwable` class to get information regarding the exception, as shown in Figure 12.4. The `printStackTrace()` method prints stack trace information on the console. The stack trace lists all the methods in the call stack, which provides

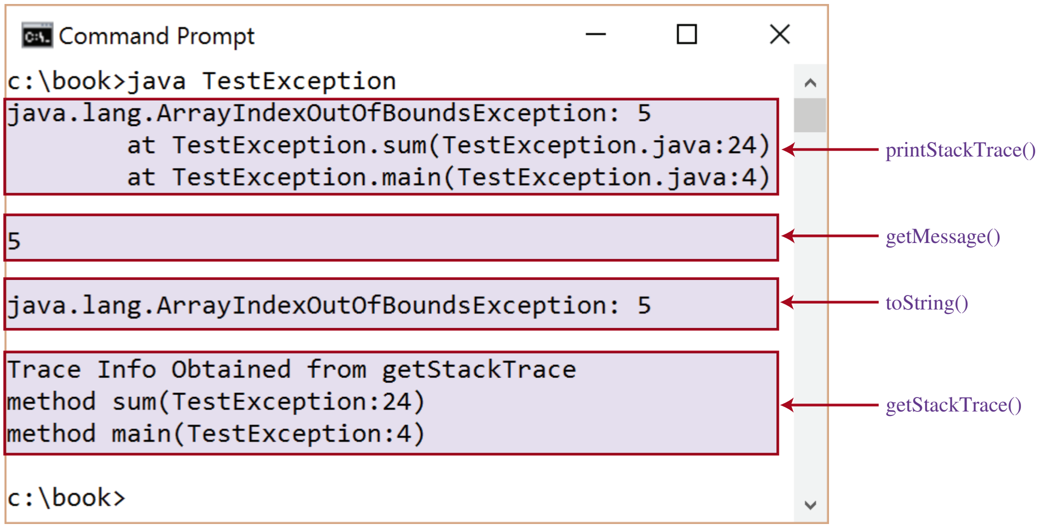


java.lang.Throwable	
`+getMessage(): String`	Returns the message that describes this exception object.
`+toString(): String`	Returns the concatenation of three strings: (1) the full name of the exception class; (2) `": "` (a colon and a space); and (3) the `getMessage()` method.
`+printStackTrace(): void`	Prints the `Throwable` object and its call stack trace information on the console.
`+getStackTrace(): StackTraceElement[]`	Returns an array of stack trace elements representing the stack trace pertaining to this exception object.

FIGURE 12.4 `Throwable` is the root class for all exception objects.

valuable information for debugging runtime errors. The `getStackTrace()` method provides programmatic access to the stack trace information printed by `printStackTrace()`.

Listing 12.6 gives an example that uses the methods in `Throwable` to display exception information. Line 4 invokes the `sum` method to return the sum of all the elements in the array. There is an error in line 23 that causes the `ArrayIndexOutOfBoundsException`, a subclass of `IndexOutOfBoundsException`. This exception is caught in the `try-catch` block. Lines 7, 8, and 9 display the stack trace, exception message, and exception object and message using the `printStackTrace()`, `getMessage()`, and `toString()` methods, as shown in Figure 12.5. Line 12 brings stack trace elements into an array. Each element represents a method call. You can obtain the method (line 14), class name (line 15), and exception line number (line 16) for each element.



```
c:\book>java TestException
java.lang.ArrayIndexOutOfBoundsException: 5
        at TestException.sum(TestException.java:24)
        at TestException.main(TestException.java:4)

5

java.lang.ArrayIndexOutOfBoundsException: 5

Trace Info Obtained from getStackTrace
method sum(TestException:24)
method main(TestException:4)

c:\book>
```

FIGURE 12.5 You can use the `printStackTrace()`, `getMessage()`, `toString()`, and `getStackTrace()` methods to obtain information from exception objects.

LISTING 12.6 TestException.java

invoke sum
printStackTrace()
getMessage()
toString()
getStackTrace()
cause an exception

```
 1  public class TestException  {
 2    public static void main(String[] args) {
 3      try {
 4        System.out.println(sum(new int[] {1, 2, 3, 4, 5}));
 5      }
 6      catch (Exception ex) {
 7        ex.printStackTrace();
 8        System.out.println("\n" + ex.getMessage());
 9        System.out.println("\n" + ex.toString());
10
11        System.out.println("\nTrace Info Obtained from getStackTrace");
12        StackTraceElement[] traceElements = ex.getStackTrace();
13        for (int i = 0; i < traceElements.length; i++) {
14          System.out.print("method " + traceElements[i].getMethodName());
15          System.out.print("(" + traceElements[i].getClassName() + ":");
16          System.out.println(traceElements[i].getLineNumber() + ")");
17        }
18      }
19    }
20
21    private static int sum(int[] list) {
22      int result = 0;
23      for (int i = 0; i <= list.length; i++)
```

```
      result += list[i];
    return result;
  }
}
```

12.4.5 Example: Declaring, Throwing, and Catching Exceptions

This example demonstrates declaring, throwing, and catching exceptions by modifying the **`setRadius`** method in the **`Circle`** class in Listing 9.8, Circle.java (CircleWithPrivateDataField). The new **`setRadius`** method throws an exception if the radius is negative.

Listing 12.7 defines a new circle class named **`CircleWithException`**, which is the same as **`Circle`** in Listing 9.8 except that the **`setRadius(double newRadius)`** method throws an **`IllegalArgumentException`** if the argument **`newRadius`** is negative.

LISTING 12.7 CircleWithException.java

```
public class CircleWithException {
  /** The radius of the circle */
  private double radius;

  /** The number of the objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithException() {
    this(1.0);
  }

  /** Construct a circle with a specified radius */
  public CircleWithException(double newRadius) {
    setRadius(newRadius);
    numberOfObjects++;
  }

  /** Return radius */
  public double getRadius() {
    return radius;
  }

  /** Set a new radius */
  public void setRadius(double newRadius)
      throws IllegalArgumentException {
    if (newRadius >= 0)
      radius = newRadius;
    else
      throw new IllegalArgumentException(
        "Radius cannot be negative");
  }

  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  public double findArea() {
    return radius * radius * 3.14159;
  }
}
```

declare exception

throw exception

A test program that uses the new `Circle` class is given in Listing 12.8.

LISTING 12.8 TestCircleWithException.java

try

catch

```
 1  public class TestCircleWithException {
 2    public static void main(String[] args) {
 3      try {
 4        CircleWithException c1 = new CircleWithException(5);
 5        CircleWithException c2 = new CircleWithException(-5);
 6        CircleWithException c3 = new CircleWithException(0);
 7      }
 8      catch (IllegalArgumentException ex) {
 9        System.out.println(ex);
10      }
11
12      System.out.println("Number of objects created: " +
13        CircleWithException.getNumberOfObjects());
14    }
15  }
```

```
java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1
```

The original `Circle` class remains intact except that the class name is changed to `CircleWithException`, a new constructor `CircleWithException(newRadius)` is added, and the `setRadius` method now declares an exception and throws it if the radius is negative.

The `setRadius` method declares to throw `IllegalArgumentException` in the method header (lines 25–32 in Listing 12.7 CircleWithException.java). The `CircleWithException` class would still compile if the `throws IllegalArgumentException` clause (line 26) were removed from the method declaration, since it is a subclass of `RuntimeException` and every method can throw `RuntimeException` (an unchecked exception) regardless of whether it is declared in the method header.

The test program creates three `CircleWithException` objects—`c1`, `c2`, and `c3`—to test how to handle exceptions. Invoking `new CircleWithException(-5)` (line 5 in Listing 12.8) causes the `setRadius` method to be invoked, which throws an `IllegalArgumentException`, because the radius is negative. In the `catch` block, the type of the object `ex` is `IllegalArgumentException`, which matches the exception object thrown by the `setRadius` method, so this exception is caught by the `catch` block.

The exception handler prints a short message, `ex.toString()` (line 9 in Listing 12.8), about the exception, using `System.out.println(ex)`.

Note that the execution continues in the event of the exception. If the handlers had not caught the exception, the program would have abruptly terminated.

The test program would still compile if the `try` statement were not used, because the method throws an instance of `IllegalArgumentException`, a subclass of `RuntimeException` (an unchecked exception).

Check Point

12.4.1 What is the purpose of declaring exceptions? How do you declare an exception and where? Can you declare multiple exceptions in a method header?

12.4.2 How do you throw an exception? Can you throw multiple exceptions in one `throw` statement?

12.4.3 What is the keyword `throw` used for? What is the keyword `throws` used for?

12.4.4 Suppose `statement2` causes an exception in the following `try-catch` block:

```
try {
  statement1;
  statement2;
  statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
}

statement4;
```

Answer the following questions:

- Will `statement3` be executed?
- If the exception is not caught, will `statement4` be executed?
- If the exception is caught in the `catch` block, will `statement4` be executed?

12.4.5 What is displayed when running the following program?

```
public class Test {
  public static void main(String[] args) {
    try {
      int[] list = new int[10];
      System.out.println("list[10] is " + list[10]);
    }
    catch (ArithmeticException ex) {
      System.out.println("ArithmeticException");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException");
    }
    catch (Exception ex) {
      System.out.println("Exception");
    }
  }
}
```

12.4.6 What is displayed when running the following program?

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (ArithmeticException ex) {
     System.out.println("ArithmeticException");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException");
    }
    catch (Exception e) {
      System.out.println("Exception");
    }
  }

  static void method() throws Exception {
     System.out.println(1 / 0);
   }
 }
```

12.4.7 What is displayed when running the following program?

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in main");
    }
    catch (Exception ex) {
      System.out.println("Exception in main");
    }
  }

  static void method() throws Exception {
    try {
      String s ="abc";
      System.out.println(s.charAt(3));
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in method()");
    }
    catch (Exception ex) {
     System.out.println("Exception in method()");
    }
  }
}
```

12.4.8 What does the method `getMessage()` do?

12.4.9 What does the method `printStackTrace()` do?

12.4.10 Does the presence of a `try-catch` block impose overhead when no exception occurs?

12.4.11 Correct a compile error in the following code:

```
public void m(int value) {
  if (value < 40)
    throw new Exception("value is too small");
}
```

12.5 The `finally` Clause

Key Point

The `finally` clause is always executed regardless of whether an exception occurred or not.

Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught. Java has a `finally` clause that can be used to accomplish this objective. The syntax for the `finally` clause might look like this:

```
try {
  statements;
}
catch (TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}
```

The code in the `finally` block is executed under all circumstances, regardless of whether an exception occurs in the `try` block or is caught. Consider three possible cases:

1. If no exception arises in the `try` block, `finalStatements` is executed and the next statement after the `try` statement is executed.
2. If a statement causes an exception in the `try` block that is caught in a `catch` block, the rest of the statements in the `try` block are skipped, the `catch` block is executed, and the `finally` clause is executed. The next statement after the `try` statement is executed.
3. If one of the statements causes an exception that is not caught in any `catch` block, the other statements in the `try` block are skipped, the `finally` clause is executed, and the exception is passed to the caller of this method.

The code in the finally clause is often for closing files and for cleaning up resources. The `finally` block executes even if there is a `return` statement prior to reaching the `finally` block.

omit catch block

Note

The `catch` block may be omitted when the `finally` clause is used, as shown in the following code:

```
try {
  code may throw a non-checked exception; regardless of whether an
  exception occurs, finalStatements are executed.
}
finally {
  finalStatements;
}
```

Check Point

12.5.1 Suppose you run the following code:

```
public static void main(String[] args) throws Exception2 {
  m();
  statement7;
}

public static void m() {
  try {
    statement1;
    statement2;
    statement3;
  }
  catch (Exception1 ex1) {
    statement4;
  }
  finally {
    statement5;
  }
  statement6;
}
```

Answer the following questions:

a. If no exception occurs, which statements are executed?

b. If `statement2` throws an exception of type `Exception1`, which statements are executed?

c. If `statement2` throws an exception of type `Exception2`, which statements are executed?

d. If `statement2` throws an exception that is neither `Exception1` nor `Exception2`, which statements are executed?

12.6 When to Use Exceptions

Key Point

A method should throw an exception if the error needs to be handled by its caller.

The `try` block contains the code that is executed in normal circumstances. The `catch` block contains the code that is executed in exceptional circumstances. Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify. Be aware, however, that exception handling usually requires more time and resources, because it requires instantiating a new exception object, rolling back the call stack, and propagating the exception through the chain of method calls to search for the handler.

An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw or use exceptions.

In general, common exceptions that may occur in multiple classes in a project are candidates for exception classes. Simple errors that may occur in individual methods are best handled without throwing exceptions. This can be done by using `if` statements to check for errors.

When should you use a `try-catch` block in the code? Use it when you have to deal with unexpected error conditions. Do not use a `try-catch` block to deal with simple, expected situations. For example, the following code:

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

is better replaced by

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

Which situations are exceptional and which are expected is sometimes difficult to decide. The point is not to abuse exception handling as a way to deal with a simple logic test.

Check Point

12.6.1 The following method checks whether a string is a numeric string:

```
public static boolean isNumeric(String token) {
  try {
    Double.parseDouble(token);
    return true;
  }
  catch (java.lang.NumberFormatException ex) {
    return false;
  }
}
```

Is it correct? Rewrite it without using exceptions.

12.7 Rethrowing Exceptions

Key Point

Java allows an exception handler to rethrow the exception if the handler cannot process the exception, or simply wants to let its caller be notified of the exception.

The syntax for rethrowing an exception may look like this:

```
try {
  statements;
}
catch (TheException ex) {
  perform operations before exits;
  throw ex;
}
```

The statement `throw ex` rethrows the exception to the caller so other handlers in the caller get a chance to process the exception `ex`.

Check Point

12.7.1 Suppose that `statement2` may cause an exception in the following code:

```
try {
  statement1;
  statement2;
  statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
  throw ex2;
}
finally {
  statement4;
}
statement5;
```

Answer the following questions:

a. If no exception occurs, will `statement4` or `statement5` be executed?

b. If the exception is of type `Exception1`, will `statement4` or `statement5` be executed?

c. If the exception is of type `Exception2`, will `statement4` or `statement5` be executed?

d. If the exception is not `Exception1` nor `Exception2`, will `statement4` or `statement5` be executed?

12.8 Chained Exceptions

Key Point

Throwing an exception along with another exception forms a chained exception.

In the preceding section, the `catch` block rethrows the original exception. Sometimes, you may need to throw a new exception (with additional information) along with the original exception. This is called *chained exceptions*. Listing 12.9 illustrates how to create and throw chained exceptions.

chained exception

Listing 12.9 ChainedExceptionDemo.java

stack trace

```
public class ChainedExceptionDemo {
  public static void main(String[] args) {
    try {
      method1();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static void method1() throws Exception {
    try {
      method2();
```

chained exception

throw exception

```
14       }
15       catch (Exception ex) {
16         throw new Exception("New info from method1", ex);
17       }
18     }
19
20     public static void method2() throws Exception {
21       throw new Exception("New info from method2");
22     }
23   }
```

```
java.lang.Exception: New info from method1
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:16)
  at ChainedExceptionDemo.main(ChainedExceptionDemo.java:4)
Caused by: java.lang.Exception: New info from method2
  at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:21)
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:13)
  ... 1 more
```

The **`main`** method invokes **`method1`** (line 4), **`method1`** invokes **`method2`** (line 13), and **`method2`** throws an exception (line 21). This exception is caught in the **`catch`** block in **`method1`** and is wrapped in a new exception in line 16. The new exception is thrown and caught in the catch block in the **`main`** method in line 6. The sample output shows the output from the **`printStackTrace()`** method in line 7. The new exception thrown from **`method1`** is displayed first, followed by the original exception thrown from **`method2`**.

Check Point

12.8.1 What would be the output if line 16 of Listing 12.9 is replaced by the following line?

```
throw new Exception("New info from method1");
```

12.9 Defining Custom Exception Classes

Key Point

You can define a custom exception class by extending the **`java.lang.Exception`** *class.*

VideoNote

Create custom exception classes

Java provides quite a few exception classes. Use them whenever possible instead of defining your own exception classes. However, if you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class, derived from **`Exception`** or from a subclass of **`Exception`**, such as **`IOException`**.

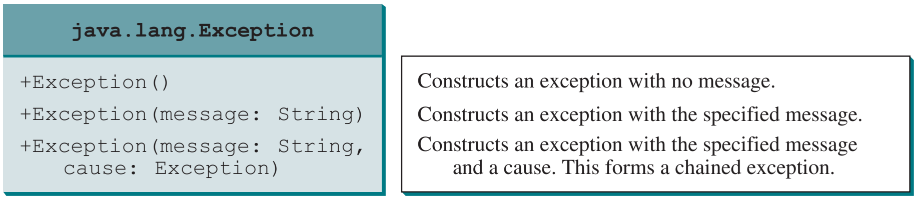
In Listing 12.7, CircleWithException.java, the **`setRadius`** method throws an exception if the radius is negative. Suppose you wish to pass the radius to the handler. In that case, you can define a custom exception class, as shown in Listing 12.10.

Listing 12.10 InvalidRadiusException.java

extends `Exception`

```
public class InvalidRadiusException extends Exception {
  private double radius;

  /** Construct an exception */
  public InvalidRadiusException(double radius) {
    super("Invalid radius " + radius);
    this.radius = radius;
  }

  /** Return the radius */
  public double getRadius() {
    return radius;
  }
}
```

This custom exception class extends `java.lang.Exception` (line 1). The `Exception` class extends `java.lang.Throwable`. All the methods (e.g., `getMessage()`, `toString()`, and `printStackTrace()`) in `Exception` are inherited from `Throwable`. The `Exception` class contains four constructors. Among them, the following constructors are often used:


java.lang.Exception
+Exception()
+Exception(message: String)
+Exception(message: String, cause: Exception)
Constructs an exception with no message.
Constructs an exception with the specified message.
Constructs an exception with the specified message and a cause. This forms a chained exception.

Line 6 invokes the superclass's constructor with a message. This message will be set in the exception object and can be obtained by invoking `getMessage()` on the object.

Tip

Most exception classes in the Java API contain two constructors: a no-arg constructor and a constructor with a message parameter.

To create an `InvalidRadiusException`, you have to pass a radius. Therefore, the `setRadius` method in Listing 12.7 can be modified as shown in Listing 12.11.

LISTING 12.11 TestCircleWithCustomException.java

```java
public class TestCircleWithCustomException {
  public static void main(String[] args) {
    try {
      new CircleWithCustomException(5);
      new CircleWithCustomException(-5);
      new CircleWithCustomException(0);
    }
    catch (InvalidRadiusException ex) {
      System.out.println(ex);
    }

    System.out.println("Number of objects created: " +
      CircleWithCustomException.getNumberOfObjects());
  }
}

class CircleWithCustomException {
  /** The radius of the circle */
  private double radius;

  /** The number of objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithCustomException() throws InvalidRadiusException {
    this(1.0);
  }

  /** Construct a circle with a specified radius */
  public CircleWithCustomException(double newRadius)
      throws InvalidRadiusException {
    setRadius(newRadius);
    numberOfObjects++;
  }

```

declare exception

throw exception

```
36     /** Return radius */
37     public double getRadius() {
38       return radius;
39     }
40
41     /** Set a new radius */
42     public void setRadius(double newRadius)
43         throws InvalidRadiusException {
44       if (newRadius >= 0)
45         radius = newRadius;
46       else
47         throw new InvalidRadiusException(newRadius);
48     }
49
50     /** Return numberOfObjects */
51     public static int getNumberOfObjects() {
52       return numberOfObjects;
53     }
54
55     /** Return the area of this circle */
56     public double findArea() {
57       return radius * radius * 3.14159;
58     }
59   }
```

```
InvalidRadiusException: Invalid radius -5.0
Number of objects created: 1
```

The `setRadius` method in `CircleWithCustomException` throws an `InvalidRadiusException` when radius is negative (line 47). Since `InvalidRadiusException` is a checked exception, the `setRadius` method must declare it in the method header (line 43). Since the constructors for `CircleWithCustomException` invoke the `setRadius` method to set a new radius, and it may throw an `InvalidRadiusException`, the constructors are declared to throw `InvalidRadiusException` (lines 25 and 31).

Invoking `new CircleWithCustomException(-5)` (line 5) throws an `InvalidRadiusException`, which is caught by the handler. The handler displays the radius in the exception object `ex`.

checked custom exception

Tip

Can you define a custom exception class by extending `RuntimeException`? Yes, but it is not a good way to go because it makes your custom exception unchecked. It is better to make a custom exception checked, so the compiler can force these exceptions to be caught in your program.

Check Point

12.9.1 How do you define a custom exception class?

12.9.2 Suppose that the `setRadius` method throws the `InvalidRadiusException` defined in Listing 12.10. What is displayed when running the following program?

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in main");
    }
    catch (Exception ex) {
      System.out.println("Exception in main");
    }
  }

  static void method() throws Exception {
    try {
      Circle c1 = new Circle(1);
      c1.setRadius(-1);
      System.out.println(c1.getRadius());
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in method()");
    }
    catch (Exception ex) {
      System.out.println("Exception in method()");
      throw ex;
    }
  }
}
```

12.10 The `File` Class

Key Point

The `File` class contains the methods for obtaining the properties of a file/directory, and for renaming and deleting a file/directory.

why file?

Having learned exception handling, you are ready to step into file processing. Data stored in the program are temporary; they are lost when the program terminates. To permanently store the data created in a program, you need to save them in a file on a disk or other permanent storage device. The file can then be transported and read later by other programs. Since data are stored in files, this section introduces how to use the `File` class to obtain file/directory properties, to delete and rename files/directories, and to create directories. The next section introduces how to read/write data from/to text files.

absolute file name

directory path

Every file is placed in a directory in the file system. An *absolute file name* (or *full name*) contains a file name with its complete path and drive letter. For example, **c:\book\Welcome.java** is the absolute file name for the file **Welcome.java** on the Windows operating system. Here, **c:\book** is referred to as the *directory path* for the file. Absolute file names are machine dependent. On the UNIX platform, the absolute file name may be **/home/liang/book/Welcome.java**, where **/home/liang/book** is the directory path for the file **Welcome.java**.

relative file name

A *relative file name* is in relation to the current working directory. The complete directory path for a relative file name is omitted. For example, **Welcome.java** is a relative file name. If the current working directory is **c:\book**, the absolute file name would be **c:\book\Welcome.java**.

The `File` class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The `File` class contains the methods for obtaining file and directory properties, and for renaming and deleting files and directories, as shown in Figure 12.6. However, *the `File` class does not contain the methods for reading and writing file contents.*

The file name is a string. The `File` class is a wrapper class for the file name and its directory path. For example, `new File("c:\\book")` creates a `File` object for the directory **c:\book** and `new File("c:\\book\\test.dat")` creates a `File` object for the file **c:\book\test.dat**, both on Windows. You can use the `File` class's `isDirectory()` method to check whether the object represents a directory, and the `isFile()` method to check whether the object represents a file.

`java.io.File`	
`+File(pathname: String)`	Creates a `File` object for the specified path name. The path name may be a directory or a file.
`+File(parent: String, child: String)`	Creates a `File` object for the child under the directory parent. The child may be a file name or a subdirectory.
`+File(parent: File, child: String)`	Creates a `File` object for the child under the directory parent. The parent is a `File` object. In the preceding constructor, the parent is a string.
`+exists(): boolean`	Returns true if the file or the directory represented by the `File` object exists.
`+canRead(): boolean`	Returns true if the file represented by the `File` object exists and can be read.
`+canWrite(): boolean`	Returns true if the file represented by the `File` object exists and can be written.
`+isDirectory(): boolean`	Returns true if the `File` object represents a directory.
`+isFile(): boolean`	Returns true if the `File` object represents a file.
`+isAbsolute(): boolean`	Returns true if the `File` object is created using an absolute path name.
`+isHidden(): boolean`	Returns true if the file represented in the `File` object is hidden. The exact definition of *hidden* is system dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period (.) character.
`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
`+getName(): String`	Returns the last name of the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getName()` returns `test.dat`.
`+getPath(): String`	Returns the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getPath()` returns `c:\book\test.dat`.
`+getParent(): String`	Returns the complete parent directory of the current directory or the file represented by the `File` object. For example, new `File("c:\\book\\test.dat").getParent()` returns `c:\book`.
`+lastModified(): long`	Returns the time that the file was last modified.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFile(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes the file or directory represented by this `File` object. The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames the file or directory represented by this `File` object to the specified name represented in dest. The method returns true if the operation succeeds.
`+mkdir(): boolean`	Creates a directory represented in this `File` object. Returns true if the the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist.

FIGURE 12.6 The `File` class can be used to obtain file and directory properties, to delete and rename files and directories, and to create directories.

\ in file names

Caution

The directory separator for Windows is a backslash (`\`). The backslash is a special character in Java and should be written as `\\` in a string literal (see Table 4.5).

Note

Constructing a **`File`** *instance does not create a file on the machine*. You can create a **`File`** instance for any file name regardless of whether it exists or not. You can invoke the **`exists()`** method on a **`File`** instance to check whether the file exists.

relative file name

Java directory separator (/)

Do not use absolute file names in your program. If you use a file name such as **`c:\\book\\Welcome.java`**, it will work on Windows but not on other platforms. You should use a file name relative to the current directory. For example, you may create a **`File`** object using **`new File("Welcome.java")`** for the file **Welcome.java** in the current directory. You may create a **`File`** object using **`new File("image/us.gif")`** for the file **us.gif** under the **image** directory in the current directory. The forward slash (**`/`**) is the Java directory separator, which is the same as on UNIX. The statement `new File("image/us.gif")` works on Windows, UNIX, and any other platform.

Listing 12.12 demonstrates how to create a `File` object and use the methods in the `File` class to obtain its properties. The program creates a `File` object for the file **us.gif**. This file is stored under the **image** directory in the current directory.

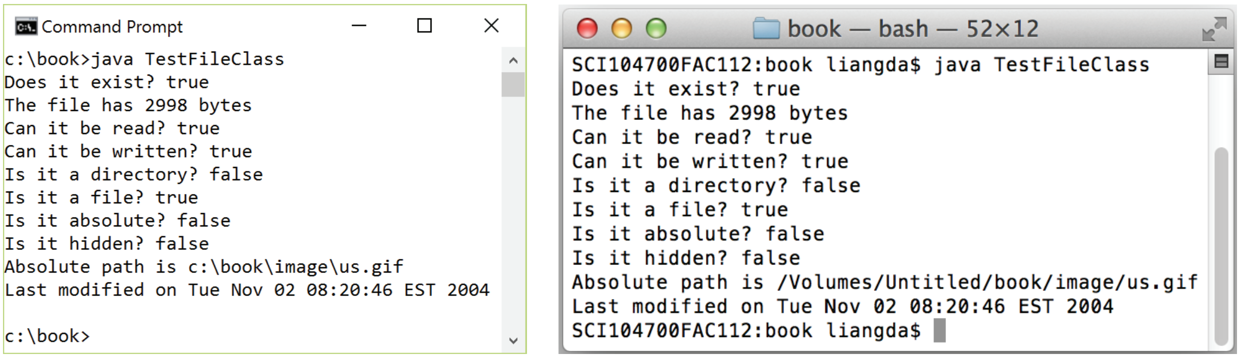
LISTING 12.12 TestFileClass.java

```
 1  public class TestFileClass {
 2    public static void main(String[] args) {
 3      java.io.File file = new java.io.File("image/us.gif");
 4      System.out.println("Does it exist? " + file.exists());
 5      System.out.println("The file has " + file.length() + " bytes");
 6      System.out.println("Can it be read? " + file.canRead());
 7      System.out.println("Can it be written? " + file.canWrite());
 8      System.out.println("Is it a directory? " + file.isDirectory());
 9      System.out.println("Is it a file? " + file.isFile());
10      System.out.println("Is it absolute? " + file.isAbsolute());
11      System.out.println("Is it hidden? " + file.isHidden());
12      System.out.println("Absolute path is " +
13        file.getAbsolutePath());
14      System.out.println("Last modified on " +
15        new java.util.Date(file.lastModified()));
16    }
17  }
```

create a `File`
`exists()`
`length()`
`canRead()`
`canWrite()`
`isDirectory()`
`isFile()`
`isAbsolute()`
`isHidden()`
`getAbsolutePath()`
`lastModified()`

The `lastModified()` method returns the date and time when the file was last modified, measured in milliseconds since the beginning of UNIX time (00:00:00 GMT, January 1, 1970). The `Date` class is used to display it in a readable format in lines 14 and 15.

Figure 12.7a shows a sample run of the program on Windows and Figure 12.7b, a sample run on UNIX. As shown in the figures, the path-naming conventions on Windows are different from those on UNIX.



```
c:\book>java TestFileClass
Does it exist? true
The file has 2998 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is c:\book\image\us.gif
Last modified on Tue Nov 02 08:20:46 EST 2004

c:\book>
```

```
SCI104700FAC112:book liangda$ java TestFileClass
Does it exist? true
The file has 2998 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is /Volumes/Untitled/book/image/us.gif
Last modified on Tue Nov 02 08:20:46 EST 2004
SCI104700FAC112:book liangda$
```

(a) On Windows

(b) On UNIX

FIGURE 12.7 The program creates a `File` object and displays file properties.

Check Point

12.10.1 What is wrong about creating a `File` object using the following statement?
`new File("c:\book\test.dat");`

12.10.2 How do you check whether a file already exists? How do you delete a file? How do you rename a file? Can you find the file size (the number of bytes) using the `File` class? How do you create a directory?

12.10.3 Can you use the `File` class for I/O? Does creating a `File` object create a file on the disk?

12.11 File Input and Output

Key Point

Use the **`Scanner`** *class for reading text data from a file, and the* **`PrintWriter`** *class for writing text data to a file.*

VideoNote
Write and read data

A **`File`** object encapsulates the properties of a file or a path, but it does not contain the methods for writing/reading data to/from a file (referred to as data *input* and *output*, or *I/O* for short). In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. There are two types of files: text and binary. Text files are essentially characters on disk. This section introduces how to read/write strings and numeric values from/to a text file using the **`Scanner`** and **`PrintWriter`** classes. Binary files will be introduced in Chapter 17.

12.11.1 Writing Data Using `PrintWriter`

The **`java.io.PrintWriter`** class can be used to create a file and write data to a text file. First, you have to create a **`PrintWriter`** object for a text file as follows:

```
PrintWriter output = new PrintWriter(filename);
```

Then, you can invoke the **`print`**, **`println`**, and **`printf`** methods on the **`PrintWriter`** object to write data to a file. Figure 12.8 summarizes frequently used methods in **`PrintWriter`**.

`java.io.PrintWriter`	
`+PrintWriter(file: File)`	Creates a `PrintWriter` object for the specified file object.
`+PrintWriter(filename: String)`	Creates a `PrintWriter` object for the specified file name string.
`+print(s: String): void`	Writes a string to the file.
`+print(c: char): void`	Writes a character to the file.
`+print(cArray: char[]): void`	Writes an array of characters to the file.
`+print(i: int): void`	Writes an `int` value to the file.
`+print(l: long): void`	Writes a `long` value to the file.
`+print(f: float): void`	Writes a `float` value to the file.
`+print(d: double): void`	Writes a `double` value to the file.
`+print(b: boolean): void`	Writes a `boolean` value to the file.
Also contains the overloaded `println` methods.	A `println` method acts like a `print` method; additionally, it prints a line separator. The line-separator string is defined by the system. It is `\r\n` on Windows and `\n` on Unix.
Also contains the overloaded `printf` methods.	The `printf` method was introduced in §4.6, “Formatting Console Output.”

FIGURE 12.8 The **`PrintWriter`** class contains the methods for writing data to a text file.

Listing 12.13 gives an example that creates an instance of **`PrintWriter`** and writes two lines to the file **scores.txt**. Each line consists of a first name (a string), a middle-name initial (a character), a last name (a string), and a score (an integer).

LISTING 12.13 `WriteData.java`

throws an exception
create `File` object
file exist?

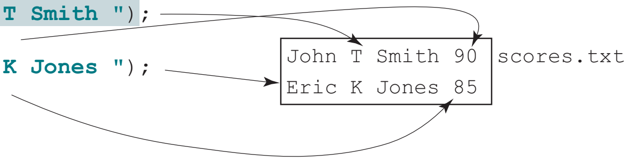
```
public class WriteData {
  public static void main(String[] args) throws java.io.IOException {
    java.io.File file = new java.io.File("scores.txt");
    if (file.exists()) {
      System.out.println("File already exists");
      System.exit(1);
    }

```

```
 9       // Create a file
10       java.io.PrintWriter output = new java.io.PrintWriter(file);
11
12       // Write formatted output to the file
13       output.print("John T Smith ");
14       output.println(90);
15       output.print("Eric K Jones ");
16       output.println(85);
17
18       // Close the file
19       output.close();
20     }
21   }
```

create `PrintWriter`

print data

close file



```
John T Smith 90
Eric K Jones 85
```

Lines 4–7 check whether the file **scores.txt** exists. If so, exit the program (line 6).

create a file

Invoking the constructor of `PrintWriter` will create a new file if the file does not exist. If the file already exists, the current content in the file will be discarded without verifying with the user.

throws `IOException`

Invoking the constructor of `PrintWriter` may throw an I/O exception. Java forces you to write the code to deal with this type of exception. For simplicity, we declare `throws IOException` in the main method header (line 2).

`print` method

You have used the `System.out.print`, `System.out.println`, and `System.out.printf` methods to write text to the console output. `System.out` is a standard Java object for the console. You can create `PrintWriter` objects for writing text to any file using `print`, `println`, and `printf` (lines 13–16).

close file

The `close()` method must be used to close the file (line 19). If this method is not invoked, the data may not be saved properly in the file.

Note

You can append data to an existing file using new PrintWriter(new FileOutputStream(file, true)) to create a PrintWriter object. FileOutputStream will be introduced in Chapter 17.

Tip

When the program writes data to a file, it first stores the data temporarily in a buffer in the memory. When the buffer is full, the data are automatically saved to the file on the disk. Once you close the file, all the data left in the buffer are saved to the file on the disk. Therefore, you must close the file to ensure that all data are saved to the file.

12.11.2 Closing Resources Automatically Using try-with-resources

Programmers often forget to close the file. JDK 7 provides the following try-with-resources syntax that automatically closes the files.

```
try (declare and create resources) {
  Use the resource to process the file;
}
```

Using the try-with-resources syntax, we rewrite the code in Listing 12.13 as shown in Listing 12.14.

LISTING 12.14 WriteDataWithAutoClose.java

```
public class WriteDataWithAutoClose {
  public static void main(String[] args) throws Exception {
    java.io.File file = new java.io.File("scores.txt");
    if (file.exists()) {
      System.out.println("File already exists");
      System.exit(0);
```

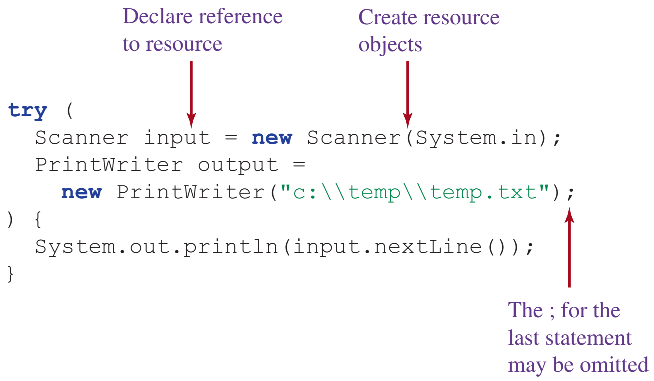
declare/create resource

use the resource

```
 7      }
 8
 9      try (
10        // Create a file
11        java.io.PrintWriter output = new java.io.PrintWriter(file);
12      ) {
13        // Write formatted output to the file
14        output.print("John T Smith ");
15        output.println(90);
16        output.print("Eric K Jones ");
17        output.println(85);
18      }
19    }
20  }
```

A resource is declared and created in the parentheses following the keyword `try`. The resources must be a subtype of `AutoCloseable` such as a `PrinterWriter` that has the `close()` method. A resource must be declared and created in the same statement, and multiple resources can be declared and created inside the parentheses. The statements in the block (lines 12–18) immediately following the resource declaration use the resource. After the block is finished, the resource's `close()` method is automatically invoked to close the resource. Using try-with-resources can not only avoid errors, but also make the code simpler. Note the catch clause may be omitted in a try-with-resources statement.

Note that (1) you have to declare the resource reference variable and create the resource altogether in the `try(...)` clause; (2) the semicolon (`;`) in last statement in the `try(...)` clause may be omitted; (3) You may create multiple `AutoCloseable` resources in the the `try(...)` clause; (4) The `try(...)` clause can contain only the statements for creating resources. Here is an example.


Declare reference to resource
Create resource objects
The ; for the last statement may be omitted

```
try (
  Scanner input = new Scanner(System.in);
  PrintWriter output =
    new PrintWriter("c:\\temp\\temp.txt");
) {
  System.out.println(input.nextLine());
}
```

12.11.3 Reading Data Using Scanner

The `java.util.Scanner` class was used to read strings and primitive values from the console in Section 2.3, Reading Input from the Console. A `Scanner` breaks its input into tokens delimited by whitespace characters. To read from the keyboard, you create a `Scanner` for `System.in`, as follows:

```
Scanner input = new Scanner(System.in);
```

To read from a file, create a `Scanner` for a file, as follows:

```
Scanner input = new Scanner(new File(filename));
```

Figure 12.9 summarizes frequently used methods in `Scanner`.

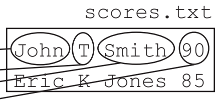
java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that produces values scanned from the specified file.
+Scanner(source: String)	Creates a Scanner that produces values scanned from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has more data to be read.
+next(): String	Returns next token as a string from this scanner.
+nextLine(): String	Returns a line ending with the line separator from this scanner.
+nextByte(): byte	Returns next token as a byte from this scanner.
+nextShort(): short	Returns next token as a short from this scanner.
+nextInt(): int	Returns next token as an int from this scanner.
+nextLong(): long	Returns next token as a long from this scanner.
+nextFloat(): float	Returns next token as a float from this scanner.
+nextDouble(): double	Returns next token as a double from this scanner.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern and returns this scanner.

FIGURE 12.9 The **Scanner** class contains the methods for scanning data.

Listing 12.15 gives an example that creates an instance of **Scanner** and reads data from the file **scores.txt**.

LISTING 12.15 ReadData.java

```
 1 import java.util.Scanner;
 2
 3 public class ReadData {
 4   public static void main(String[] args) throws Exception {
 5     // Create a File instance
 6     java.io.File file = new java.io.File("scores.txt");
 7
 8     // Create a Scanner for the file
 9     Scanner input = new Scanner(file);
10
11     // Read data from a file
12     while (input.hasNext()) {
13       String firstName = input.next();
14       String mi = input.next();
15       String lastName = input.next();
16       int score = input.nextInt();
17       System.out.println(
18         firstName +" " + mi + " " + lastName + " " + score);
19     }
20
21     // Close the file
22     input.close();
23   }
24 }
```



create a File

create a Scanner

has next?
read items

close file

Note **new Scanner(String)** creates a **Scanner** for a given string. To create a **Scanner** to read data from a file, you have to use the **java.io.File** class to create an instance of the **File** using the constructor **new File(filename)** (line 6) and use **new Scanner(File)** to create a **Scanner** for the file (line 9).

File class

Invoking the constructor **new Scanner(File)** may throw an I/O exception, so the **main** method declares **throws Exception** in line 4.

throws Exception

Each iteration in the **while** loop reads the first name, middle initial, last name, and score from the text file (lines 12–19). The file is closed in line 22.

close file

It is not necessary to close the input file (line 22), but it is a good practice to do so to release the resources occupied by the file. You can rewrite this program using the try-with-resources syntax. See liveexample.pearsoncmg.com/html/ReadDataWithAutoClose.html.

12.11.4 How Does **Scanner** Work?

change delimiter

Section 4.5.5 introduced token-based and line-based input. The token-based input methods `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, and `next()` read input separated by delimiters. By default, the delimiters are whitespace characters. You can use the `useDelimiter(String regex)` method to set a new pattern for delimiters.

How does an input method work? A token-based input first skips any delimiters (whitespace characters by default) then reads a token ending at a delimiter. The token is then automatically converted into a value of the `byte`, `short`, `int`, `long`, `float`, or `double` type for `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, and `nextDouble()`, respectively. For the `next()` method, no conversion is performed. If the token does not match the expected type, a runtime exception `java.util.InputMismatchException` will be thrown.

`InputMismatchException`

`next()` vs. `nextLine()`

Both methods `next()` and `nextLine()` read a string. The `next()` method reads a string separated by delimiters and `nextLine()` reads a line ending with a line separator.

line separator

Note

The line-separator string is defined by the system. It is `\r\n` on Windows and `\n` on UNIX. To get the line separator on a particular platform, use

```
String lineSeparator = System.getProperty("line.separator");
```

If you enter input from a keyboard, a line ends with the *Enter* key, which corresponds to the `\n` character.

behavior of `nextLine()`

The token-based input method does not read the delimiter after the token. If the `nextLine()` method is invoked after a token-based input method, this method reads characters that start from this delimiter and end with the line separator. The line separator is read, but it is not part of the string returned by `nextLine()`.

input from file

Suppose a text file named **test.txt** contains a line

```
34 567
```

After the following code is executed,

```
Scanner input = new Scanner(new File("test.txt"));
int intValue = input.nextInt();
String line = input.nextLine();
```

`intValue` contains `34` and `line` contains the characters `' '`, `5`, `6`, and `7`.

What happens if the input is *entered from the keyboard*? Suppose you enter `34`, press the *Enter* key, then enter `567` and press the *Enter* key for the following code:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
String line = input.nextLine();
```

You will get `34` in `intValue` and an empty string in `line`. Why? Here is the reason. The token-based input method `nextInt()` reads in `34` and stops at the delimiter, which in this case is a line separator (the *Enter* key). The `nextLine()` method ends after reading the line separator and returns the string read before the line separator. Since there are no characters before the line separator, `line` is empty. For this reason, *you should not use a line-based input after a token-based input.*

scan a string

You can read data from a file or from the keyboard using the `Scanner` class. You can also scan data from a string using the `Scanner` class. For example, the following code:

```
Scanner input = new Scanner("13 14");
int sum = input.nextInt() + input.nextInt();
System.out.println("Sum is " + sum);
```

displays

```
Sum is 27
```

12.11.5 Case Study: Replacing Text

Suppose you are to write a program named `ReplaceText` that replaces all occurrences of a string in a text file with a new string. The file name and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString newString
```

For example, invoking

```
java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
```

replaces all the occurrences of `StringBuilder` by `StringBuffer` in the file **FormatString.java** and saves the new file in **t.txt**.

Listing 12.16 gives the program. The program checks the number of arguments passed to the `main` method (lines 7–11), checks whether the source and target files exist (lines 14–25), creates a `Scanner` for the source file (line 29), creates a `PrintWriter` for the target file (line 30), and repeatedly reads a line from the source file (line 33), replaces the text (line 34), and writes a new line to the target file (line 35).

LISTING 12.16 `ReplaceText.java`

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class ReplaceText {
 5    public static void main(String[] args) throws Exception {
 6      // Check command line parameter usage
 7      if (args.length != 4) {
 8        System.out.println(
 9          "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
10        System.exit(1);
11      }
12
13      // Check if source file exists
14      File sourceFile = new File(args[0]);
15      if (!sourceFile.exists()) {
16        System.out.println("Source file " + args[0] + " does not exist");
17        System.exit(2);
18      }
19
20      // Check if target file exists
21      File targetFile = new File(args[1]);
22      if (targetFile.exists()) {
23        System.out.println("Target file " + args[1] + " already exists");
24        System.exit(3);
25      }
26
27      try (
28        // Create input and output files
29        Scanner input = new Scanner(sourceFile);
30        PrintWriter output = new PrintWriter(targetFile);
```

check command usage

source file exists?

target file exists?

try-with-resources

create a `Scanner`

create a `PrintWriter`

has next?
read a line

```
31        ) {
32          while (input.hasNext()) {
33            String s1 = input.nextLine();
34            String s2 = s1.replaceAll(args[2], args[3]);
35            output.println(s2);
36          }
37        }
38      }
39    }
```

In a normal situation, the program is terminated after a file is copied. The program is terminated abnormally if the command-line arguments are not used properly (lines 7–11), if the source file does not exist (lines 14–18), or if the target file already exists (lines 22–25). The exit status codes 1, 2, and 3 are used to indicate these abnormal terminations (lines 10, 17, and 24).

Check Point

12.11.1 How do you create a `PrintWriter` to write data to a file? What is the reason to declare `throws Exception` in the main method in Listing 12.13, WriteData.java? What would happen if the `close()` method were not invoked in Listing 12.13?

12.11.2 Show the contents of the file **temp.txt** after the following program is executed:

```
public class Test {
  public static void main(String[] args) throws Exception {
    java.io.PrintWriter output = new
      java.io.PrintWriter("temp.txt");
    output.printf("amount is %f %e\r\n", 32.32, 32.32);
    output.printf("amount is %5.4f %5.4e\r\n", 32.32, 32.32);
    output.printf("%6b\r\n", (1 > 2));
    output.printf("%6s\r\n", "Java");
    output.close();
  }
}
```

12.11.3 Rewrite the code in the preceding question using a try-with-resources syntax.

12.11.4 How do you create a `Scanner` to read data from a file? What is the reason to define `throws Exception` in the main method in Listing 12.15, ReadData.java? What would happen if the `close()` method were not invoked in Listing 12.15?

12.11.5 What will happen if you attempt to create a `Scanner` for a nonexistent file? What will happen if you attempt to create a `PrintWriter` for an existing file?

12.11.6 Is the line separator the same on all platforms? What is the line separator on Windows?

12.11.7 Suppose you enter `45 57.8 789`, then press the *Enter* key. Show the contents of the variables after the following code is executed:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

12.11.8 Suppose you enter `45`, press the *Enter* key, enter `57.8`, press the *Enter* key, and enter `789`, press the *Enter* key. Show the contents of the variables after the following code is executed:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

12.12 Reading Data from the Web

Just like you can read data from a file on your computer, you can read data from a file on the Web.

Key Point

In addition to reading data from a local file on a computer or file server, you can also access data from a file that is on the Web if you know the file's URL (Uniform Resource Locator—the unique address for a file on the Web). For example, www.google.com/index.html is the URL for the file **index.html** located on the Google web server. When you enter the URL in a Web browser, the Web server sends the data to your browser, which renders the data graphically. Figure 12.10 illustrates how this process works.

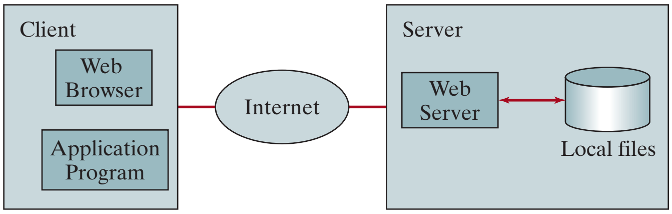


FIGURE 12.10 The client retrieves files from a Web server.

For an application program to read data from a URL, you first need to create a `URL` object using the `java.net.URL` class with this constructor:

```
public URL(String spec) throws MalformedURLException
```

For example, the following statement creates a URL object for http://www.google.com/index.html.

```
1  try {
2    URL url = new URL("http://www.google.com/index.html");
3  }
4  catch (MalformedURLException ex) {
5    ex.printStackTrace();
6  }
```

A `MalformedURLException` is thrown if the URL string has a syntax error. For example, the URL string http:www.google.com/index.html would cause a `MalformedURLException` runtime error because two slashes (`//`) are required after the colon (`:`). Note the `http://` prefix is required for the `URL` class to recognize a valid URL. It would be wrong if you replace line 2 with the following code:

```
URL url = new URL("www.google.com/index.html");
```

After a `URL` object is created, you can use the `openStream()` method defined in the `URL` class to open an input stream and use this stream to create a `Scanner` object as follows:

```
Scanner input = new Scanner(url.openStream());
```

Now you can read the data from the input stream just like from a local file. The example in Listing 12.17 prompts the user to enter a URL and displays the size of the file.

LISTING 12.17 ReadFileFromURL.java

```
import java.util.Scanner;

public class ReadFileFromURL {
  public static void main(String[] args) {
```

enter a URL

create a URL object

create a Scanner object

more to read?

read a line

`MalformedURLException`

`IOException`

```
 5        System.out.print("Enter a URL: ");
 6        String URLString = new Scanner(System.in).next();
 7
 8        try {
 9          java.net.URL url = new java.net.URL(URLString);
10          int count = 0;
11          Scanner input = new Scanner(url.openStream());
12          while (input.hasNext()) {
13            String line = input.nextLine();
14            count += line.length();
15          }
16
17         System.out.println("The file size is " + count + " characters");
18        }
19        catch (java.net.MalformedURLException ex) {
20          System.out.println("Invalid URL");
21        }
22        catch (java.io.IOException ex) {
23          System.out.println("I/O Errors: no such file");
24        }
25      }
26    }
```

```
Enter a URL: http://liveexample.pearsoncmg.com/data/Lincoln.txt ↵ Enter
The file size is 1469 characters
```

```
Enter a URL: http://www.yahoo.com ↵ Enter
The file size is 190006 characters
```

`MalformedURLException`

The program prompts the user to enter a URL string (line 6) and creates a **`URL`** object (line 9). The constructor will throw a **`java.net.MalformedURLException`** (line 19) if the URL isn't formed correctly.

The program creates a **`Scanner`** object from the input stream for the URL (line 11). If the URL is formed correctly but does not exist, an **`IOException`** will be thrown (line 22). For example, http://google.com/index1.html uses the appropriate form, but the URL itself does not exist. An **`IOException`** would be thrown if this URL was used for this program.

Check Point

12.12.1 How do you create a **`Scanner`** object for reading text from a URL?

12.13 Case Study: Web Crawler

Key Point

This case study develops a program that travels the Web by following hyperlinks.

web crawler

The World Wide web, abbreviated as WWW, W3, or Web, is a system of interlinked hypertext documents on the Internet. With a web browser, you can view a document and follow the hyperlinks to view other documents. In this case study, we will develop a program that automatically traverses the documents on the Web by following the hyperlinks. This type of program is commonly known as a *web crawler*. For simplicity, our program follows the hyperlink that starts with **`http://`**. Figure 12.11 shows an example of traversing the Web. We start from a Webpage that contains three URLs named **`URL1`**, **`URL2`**, and **`URL3`**. Following **`URL1`** leads to the page that contains three URLs named **`URL11`**, **`URL12`**, and **`URL13`**. Following **`URL2`** leads to the page that contains two URLs named **`URL21`** and **`URL22`**. Following **`URL3`** leads to the page that contains four URLs named **`URL31`**, **`URL32`**, **`URL33`**, and **`URL34`**. Continue to traverse

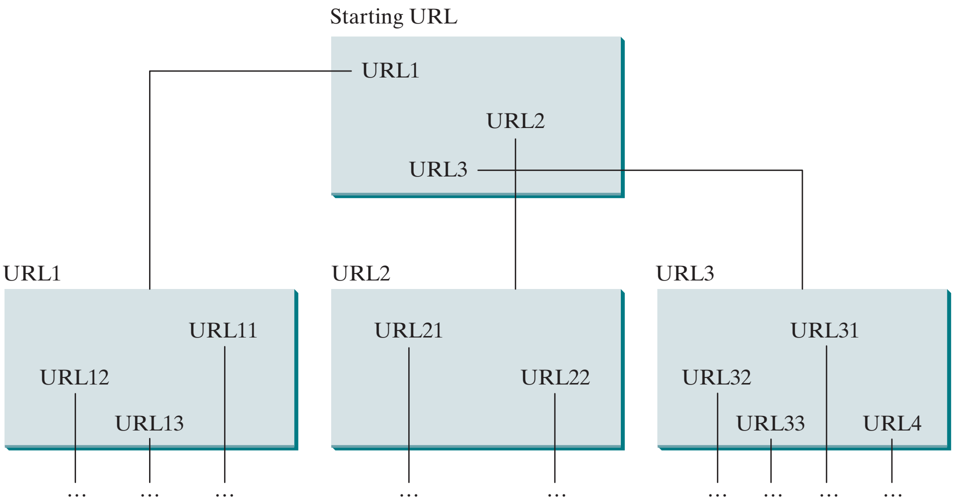


FIGURE 12.11 Web crawler explores the web through hyperlinks.

the Web following the new hyperlinks. As you see, this process may continue forever, but we will exit the program once we have traversed 100 pages.

The program follows the URLs to traverse the Web. To ensure that each URL is traversed only once, the program maintains two lists of URLs. One list stores the URLs pending for traversing, and the other stores the URLs that have already been traversed. The algorithm for this program can be described as follows:

```
Add the starting URL to a list named listOfPendingURLs;
while listOfPendingURLs is not empty and size of listOfTraversedURLs
<= 100 {
  Remove a URL from listOfPendingURLs;
  if this URL is not in listOfTraversedURLs {
    Add it to listOfTraversedURLs;
    Display this URL;
    Read the page from this URL and for each URL contained in the page {
      Add it to listOfPendingURLs if it is not in listOfTraversedURLs;
    }
  }
}
```

Listing 12.18 gives the program that implements this algorithm.

LISTING 12.18 WebCrawler.java

```java
import java.util.Scanner;
import java.util.ArrayList;

public class WebCrawler {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a URL: ");
    String url = input.nextLine();
    crawler(url); // Traverse the Web from the a starting url
  }

  public static void crawler(String startingURL) {
    ArrayList<String> listOfPendingURLs = new ArrayList<>();
    ArrayList<String> listOfTraversedURLs = new ArrayList<>();
```

enter a URL

crawl from this URL

list of pending URLs

list of traversed URLs

add starting URL

get the first URL

URL traversed

add a new URL

read a line

search for a URL

end of a URL

URL ends with "

extract a URL

search for next URL

return URLs

```
15
16       listOfPendingURLs.add(startingURL);
17       while (!listOfPendingURLs.isEmpty() &&
18           listOfTraversedURLs.size() <= 100) {
19         String urlString = listOfPendingURLs.remove(0);
20         if (!listOfTraversedURLs.contains(urlString)) {
21           listOfTraversedURLs.add(urlString);
22           System.out.println("Crawl " + urlString);
23
24           for (String s: getSubURLs(urlString)) {
25             if (!listOfTraversedURLs.contains(s))
26               listOfPendingURLs.add(s);
27           }
28         }
29       }
30     }
31
32     public static ArrayList<String> getSubURLs(String urlString) {
33       ArrayList<String> list = new ArrayList<>();
34
35       try {
36         java.net.URL url = new java.net.URL(urlString);
37         Scanner input = new Scanner(url.openStream());
38         int current = 0;
39         while (input.hasNext()) {
40           String line = input.nextLine();
41           current = line.indexOf("http:", current);
42           while (current > 0) {
43             int endIndex = line.indexOf("\"", current);
44             if (endIndex > 0) { // Ensure that a correct URL is found
45               list.add(line.substring(current, endIndex));
46               current = line.indexOf("http:", endIndex);
47             }
48             else
49               current = -1;
50           }
51         }
52       }
53       catch (Exception ex) {
54         System.out.println("Error: " + ex.getMessage());
55       }
56
57       return list;
58     }
59   }
```

```
Enter a URL: http://cs.armstrong.edu/liang ↵ Enter
Crawl http://www.cs.armstrong.edu/liang
Crawl http://www.cs.armstrong.edu
Crawl http://www.armstrong.edu
Crawl http://www.pearsonhighered.com/liang
...
```

The program prompts the user to enter a starting URL (lines 7 and 8) and invokes the `crawler(url)` method to traverse the Web (line 9).

The `crawler(url)` method adds the starting url to `listOfPendingURLs` (line 16) and repeatedly process each URL in `listOfPendingURLs` in a while loop (lines 17–29). It removes the first URL in the list (line 19) and processes the URL if it has not been processed (lines 20–28).

To process each URL, the program first adds the URL to `listOfTraversedURLs` (line 21). This list stores all the URLs that have been processed. The `getSubURLs(url)` method returns a list of URLs in the webpage for the specified URL (line 24). The program uses a foreach loop to add each URL in the page into `listOfPendingURLs` if it is not in `listOfTraversedURLs` (lines 24–27).

The `getSubURLs(url)` method reads each line from the webpage (line 40) and searches for the URLs in the line (line 41). Note a correct URL cannot contain line break characters. Therefore, it is sufficient to limit the search for a URL in one line of the text in a webpage. For simplicity, we assume that a URL ends with a quotation mark `"` (line 43). The method obtains a URL and adds it to a list (line 45). A line may contain multiple URLs. The method continues to search for the next URL (line 46). If no URL is found in the line, current is set to `-1` (line 49). The URLs contained in the page are returned in the form of a list (line 57).

The program terminates when the number of traversed URLs reaches 100 (line 18).

This is a simple program to traverse the Web. Later, you will learn the techniques to make the program more efficient and robust.

Check Point

12.13.1 Before a URL is added to `listOfPendingURLs`, line 25 checks whether it has been traversed. Is it possible that `listOfPendingURLs` contains duplicate URLs? If so, give an example.

12.13.2 Simplify the code in lines 20-28 as follows: 1. Delete lines 20 and 28; 2. Add an additional condition `!listOfPendingURLs.contains(s)` to the if statement in line 25. Write the complete new code for the while loop in lines 17-29. Does this revision work?

KEY TERMS

absolute file name	477
chained exception	473
checked exception	461
declare exception	467
directory path	477
exception	491
exception propagation	463
relative file name	477
throw exception	457
unchecked exception	489

CHAPTER SUMMARY

1. Exception handling enables a method to throw an exception to its caller.

2. A Java *exception* is an instance of a class derived from `java.lang.Throwable`. Java provides a number of predefined exception classes, such as `Error`, `Exception`, `RuntimeException`, `ClassNotFoundException`, `NullPointerException`, and `ArithmeticException`. You can also define your own exception class by extending `Exception`.

3. Exceptions occur during the execution of a method. `RuntimeException` and `Error` are *unchecked exceptions*; all other exceptions are *checked*.

4. When *declaring a method*, you have to declare a checked exception if the method might throw it, thus telling the compiler what can go wrong.

5. The keyword for declaring an exception is `throws`, and the keyword for throwing an exception is `throw`.

6. To invoke the method that declares checked exceptions, enclose it in a `try` statement. When an exception occurs during the execution of the method, the `catch` block catches and handles the exception.

7. If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

8. Various exception classes can be derived from a common superclass. If a `catch` block catches the exception objects of a superclass, it can also catch all the exception objects of the subclasses of that superclass.

9. The order in which exceptions are specified in a `catch` block is important. A compile error will result if you specify an exception object of a class after an exception object of the superclass of that class.

10. When an exception occurs in a method, the method exits immediately if it does not catch the exception. If the method is required to perform some task before exiting, you can catch the exception in the method and then rethrow it to its caller.

11. The code in the `finally` block is executed under all circumstances, regardless of whether an exception occurs in the `try` block, or whether an exception is caught if it occurs.

12. Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.

13. Exception handling should not be used to replace simple tests. You should perform simple test using `if` statements whenever possible and reserve exception handling for dealing with situations that cannot be handled with `if` statements.

14. The `File` class is used to obtain file properties and manipulate files. It does not contain the methods for creating a file or for reading/writing data from/to a file.

15. You can use `Scanner` to read string and primitive data values from a text file and use `PrintWriter` to create a file and write data to a text file.

16. You can read from a file on the Web using the `URL` class.

QUIZ

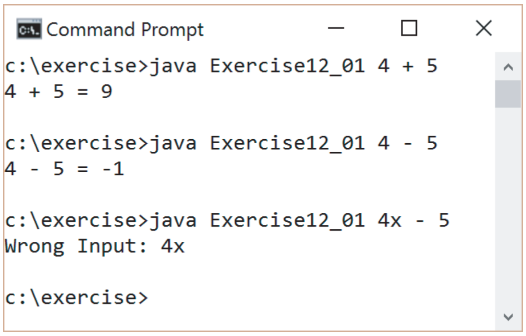
Answer the quiz for this chapter online at the Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

Sections 12.2–12.9

***12.1** (*`NumberFormatException`*) Listing 7.9, Calculator.java, is a simple command-line calculator. Note the program terminates if any operand is nonnumeric. Write a program with an exception handler that deals with nonnumeric operands; then write another program without using an exception handler to achieve the same objective. Your program should display a message that informs the user of the wrong operand type before exiting (see Figure 12.12).



```
c:\exercise>java Exercise12_01 4 + 5
4 + 5 = 9

c:\exercise>java Exercise12_01 4 - 5
4 - 5 = -1

c:\exercise>java Exercise12_01 4x - 5
Wrong Input: 4x

c:\exercise>
```

FIGURE 12.12 The program performs arithmetic operations and detects input errors.

12.2** (InputMismatchException***) Write a program that prompts the user to read two integers and displays their sum. Your program should prompt the user to read the number again if the input is incorrect.

12.3** (ArrayIndexOutOfBoundsException***) Write a program that meets the following requirements:

- Creates an array with `100` randomly chosen integers.
- Prompts the user to enter the index of the array, then displays the corresponding element value. If the specified index is out of bounds, display the message "Out of Bounds".

12.4** (IllegalArgumentException***) Modify the `Loan` class in Listing 10.2 to throw `IllegalArgumentException` if the loan amount, interest rate, or number of years is less than or equal to zero.

12.5** (IllegalTriangleException***) Programming Exercise 11.1 defined the `Triangle` class with three sides. In a triangle, the sum of any two sides is greater than the other side. The `Triangle` class must adhere to this rule. Create the `IllegalTriangleException` class, and modify the constructor of the `Triangle` class to throw an `IllegalTriangleException` object if a triangle is created with sides that violate the rule, as follows:

```
/** Construct a triangle with the specified sides */
public Triangle(double side1, double side2, double side3)
  throws IllegalTriangleException {
  // Implement it
}
```

12.6** (NumberFormatException***) Listing 6.8 implements the `hex2Dec(String hexString)` method, which converts a hex string into a decimal number. Implement the `hex2Dec` method to throw a `NumberFormatException` if the string is not a hex string. Write a test program that prompts the user to enter a hex number as a string and displays its decimal equivalent. If the method throws an exception, display “Not a hex number”.

12.7** (NumberFormatException***) Write the `bin2Dec(String binaryString)` method to convert a binary string into a decimal number. Implement the `bin2Dec` method to throw a `NumberFormatException` if the string is not a binary string. Write a test program that prompts the user to enter a binary number as a string and displays its decimal equivalent. If the method throws an exception, display “Not a binary number”.

12.8** (HexFormatException***) Programming Exercise 12.6 implements the `hex2Dec` method to throw a `NumberFormatException` if the string is not a hex string. Define a custom exception called `HexFormatException`.

VideoNote
HexFormatException

Implement the `hex2Dec` method to throw a `HexFormatException` if the string is not a hex string.

12.9 (`BinaryFormatException`*) Exercise 12.7 implements the `bin2Dec` method to throw a `BinaryFormatException` if the string is not a binary string. Define a custom exception called `BinaryFormatException`. Implement the `bin2Dec` method to throw a `BinaryFormatException` if the string is not a binary string.

12.10 (`OutOfMemoryError`*) Write a program that causes the JVM to throw an `OutOfMemoryError` and catches and handles this error.

Sections 12.10–12.12

**12.11 (*Remove text*) Write a program that removes all the occurrences of a specified string from a text file. For example, invoking

```
java Exercise12_11 John filename
```

removes the string `John` from the specified file. Your program should get the arguments from the command line.

**12.12 (*Reformat Java source code*) Write a program that converts the Java source code from the next-line brace style to the end-of-line brace style. For example, the following Java source in (a) uses the next-line brace style. Your program converts it to the end-of-line brace style in (b).

```
public class Test
{
  public static void main(String[] args)
  {
    // Some statements
  }
}
```

(a) Next-line brace style

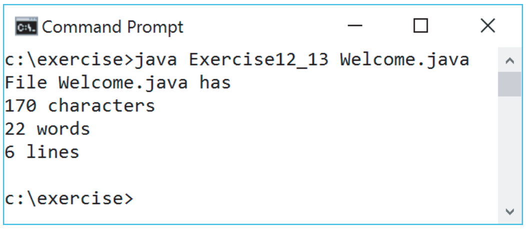
```
public class Test {
  public static void main(String[] args) {
    // Some statements
  }
}
```

(b) End-of-line brace style

Your program can be invoked from the command line with the Java source-code file as the argument. It converts the Java source code to a new format. For example, the following command converts the Java source-code file **Test.java** to the end-of-line brace style.

```
java Exercise12_12 Test.java
```

*12.13 (*Count characters, words, and lines in a file*) Write a program that will count the number of characters, words, and lines in a file. Words are separated by whitespace characters. The file name should be passed as a command-line argument, as shown in Figure 12.13.



```
c:\exercise>java Exercise12_13 Welcome.java
File Welcome.java has
170 characters
22 words
6 lines

c:\exercise>
```

FIGURE 12.13 The program displays the number of characters, words, and lines in the given file.

*12.14 (*Process scores in a text file*) Suppose a text file contains an unspecified number of scores separated by spaces. Write a program that prompts the user to enter the file, reads the scores from the file, and displays their total and average.

***12.15** (*Write/read data*) Write a program to create a file named **Exercise12_15.txt** if it does not exist. Write `100` integers created randomly into the file using text I/O. Integers are separated by spaces in the file. Read the data back from the file and display the data in increasing order.

****12.16** (*Replace text*) Listing 12.16, ReplaceText.java, gives a program that replaces text in a source file and saves the change into a new file. Revise the program to save the change into the original file. For example, invoking

```
java Exercise12_16 file oldString newString
```

replaces `oldString` in the source file with `newString`.

*****12.17** (*Game: hangman*) Rewrite Programming Exercise 7.35. The program reads the words stored in a text file named **hangman.txt**. Words are delimited by spaces.

****12.18** (*Add package statement*) Suppose you have Java source files under the directories `chapter1`, `chapter2`, . . . , `chapter34`. Write a program to insert the statement `package chapteri;` as the first line for each Java source file under the directory `chapteri`. Suppose `chapter1`, `chapter2`, . . . , `chapter34` are under the root directory `srcRootDirectory`. The root directory and `chapteri` directory may contain other folders and files. Use the following command to run the program:

```
java Exercise12_18 srcRootDirectory
```

***12.19** (*Count words*) Write a program that counts the number of words in President Abraham Lincoln's Gettysburg address from https://liveexample.pearsoncmg.com/data/Lincoln.txt.

****12.20** (*Remove package statement*) Suppose you have Java source files under the directories `chapter1`, `chapter2`, . . . , `chapter34`. Write a program to remove the statement `package chapteri;` in the first line for each Java source file under the directory `chapteri`. Suppose `chapter1`, `chapter2`, . . . , `chapter34` are under the root directory `srcRootDirectory`. The root directory and `chapteri` directory may contain other folders and files. Use the following command to run the program:

```
java Exercise12_20 srcRootDirectory
```

***12.21** (*Data sorted?*) Write a program that reads the strings from file **SortedStrings.txt** and reports whether the strings in the files are stored in increasing order. If the strings are not sorted in the file, it displays the first two strings that are out of the order.

****12.22** (*Replace text*) Revise Programming Exercise 12.16 to replace a string in a file with a new string for all files in the specified directory using the following command:

```
java Exercise12_22 dir oldString newString
```

****12.23** (*Process scores in a text file on the Web*) Suppose the text file on the Web http://liveexample.pearsoncmg.com/data/Scores.txt contains an unspecified number of scores separated by spaces. Write a program that reads the scores from the file and displays their total and average.

***12.24** (*Create large dataset*) Create a data file with 1,000 lines. Each line in the file consists of a faculty member's first name, last name, rank, and salary. The faculty member's first name and last name for the *i*th line are FirstName*i* and LastName*i*. The rank is randomly generated as assistant, associate, and full. The salary is randomly generated as a number with two digits after the decimal point. The salary for an assistant professor should be in the range from 50,000 to 80,000, for associate professor from 60,000 to 110,000, and for full professor from 75,000 to 130,000. Save the file in **Salary.txt**. Here are some sample data:

FirstName1 LastName1 assistant 60055.95

FirstName2 LastName2 associate 81112.45

. . .

FirstName1000 LastName1000 full 92255.21

***12.25** (*Process large dataset*) A university posts its employees' salaries at http://liveexample.pearsoncmg.com/data/Salary.txt. Each line in the file consists of a faculty member's first name, last name, rank, and salary (see Programming Exercise 12.24). Write a program to display the total salary for assistant professors, associate professors, full professors, and faculty, respectively, and display the average salary for assistant professors, associate professors, full professors, and faculty, respectively.

****12.26** (*Create a directory*) Write a program that prompts the user to enter a directory name and creates a directory using the `File`'s `mkdirs` method. The program displays the message "Directory created successfully" if a directory is created or "Directory already exists" if the directory already exists.

****12.27** (*Replace words*) Suppose you have a lot of files in a directory that contain words **Exercise*i*_*j***, where *i* and *j* are digits. Write a program that pads a 0 before *i* if *i* is a single digit and 0 before *j* if *j* is a single digit. For example, the word **Exercise2_1** in a file will be replaced by **Exercise02_01**. In Java, when you pass the symbol `*` from the command line, it refers to all files in the directory (see Supplement III.V). Use the following command to run your program:

```
java Exercise12_27 *
```

****12.28** (*Rename files*) Suppose you have a lot of files in a directory named **Exercise*i*_*j***, where *i* and *j* are digits. Write a program that pads a 0 before *i* if *i* is a single digit. For example, a file named **Exercise2_1** in a directory will be renamed to **Exercise02_1**. In Java, when you pass the symbol `*` from the command line, it refers to all files in the directory (see Supplement III.V). Use the following command to run your program:

```
java Exercise12_28 *
```

****12.29** (*Rename files*) Suppose you have several files in a directory named **Exercise*i*_*j***, where *i* and *j* are digits. Write a program that pads a 0 before *j* if *j* is a single digit. For example, a file named **Exercise2_1** in a directory will be renamed to **Exercise2_01**. In Java, when you pass the symbol `*` from the command line, it refers to all files in the directory (see Supplement III.V). Use the following command to run your program:

```
java Exercise12_29 *
```

****12.30** (*Occurrences of each letter*) Write a program that prompts the user to enter a file name and displays the occurrences of each letter in the file. Letters are case insensitive. Here is a sample run:

```
Enter a filename: Lincoln.txt ↵Enter
Number of As: 56
Number of Bs: 134
...
Number of Zs: 9
```

***12.31** (*Baby name popularity ranking*) The popularity ranking of baby names from years 2001 to 2010 is downloaded from www.ssa.gov/oact/babynames and stored in files named **babynameranking2001.txt**, **babynameranking2002.txt**, . . . , **babynameranking2010.txt**. You can download these files using the URL such as http://liveexample.pearsoncmg.com/data/babynamesranking2001.txt. Each file contains 1,000 lines. Each line contains a ranking, a boy's name, number for the boy's name, a girl's name, and number for the girl's name. For example, the first two lines in the file **babynameranking2010.txt** are as follows:

```
1       Jacob      21,875       Isabella       22,731
2       Ethan      17,866       Sophia         20,477
```

Therefore, the boy's name Jacob and girl's name Isabella are ranked #1 and the boy's name Ethan and girl's name Sophia are ranked #2; 21,875 boys are named Jacob, and 22,731 girls are named Isabella. Write a program that prompts the user to enter the year, gender, followed by a name, and displays the ranking of the name for the year. Your program should read the data directly from the Web. Here are some sample runs:

```
Enter the year: 2010 ↵Enter
Enter the gender: M ↵Enter
Enter the name: Javier ↵Enter
Javier is ranked #190 in year 2010
```

```
Enter the year: 2010 ↵Enter
Enter the gender: F ↵Enter
Enter the name: ABC ↵Enter
The name ABC is not ranked in year 2010
```

***12.32** (*Ranking summary*) Write a program that uses the files described in Programming Exercise 12.31 and displays a ranking summary table for the first five girl's and boy's names as follows:

```
Year Rank 1    Rank 2    Rank 3    Rank 4    Rank 5    Rank 1   Rank 2    Rank 3    Rank 4    Rank 5
2010 Isabella  Sophia    Emma      Olivia    Ava       Jacob    Ethan     Michael   Jayden    William
2009 Isabella  Emma      Olivia    Sophia    Ava       Jacob    Ethan     Michael   Alexander William
2008 Emma      Isabella  Emily     Olivia    Ava       Jacob    Michael   Ethan     Joshua    Daniel
2007 Emily     Isabella  Emma      Ava       Madison   Jacob    Michael   Ethan     Joshua    Daniel
2006 Emily     Emma      Madison   Isabella  Ava       Jacob    Michael   Joshua    Ethan     Matthew
2005 Emily     Emma      Madison   Abigail   Olivia    Jacob    Michael   Joshua    Matthew   Ethan
2004 Emily     Emma      Madison   Olivia    Hannah    Jacob    Michael   Joshua    Matthew   Ethan
2003 Emily     Emma      Madison   Hannah    Olivia    Jacob    Michael   Joshua    Matthew   Andrew
2002 Emily     Madison   Hannah    Emma      Alexis    Jacob    Michael   Joshua    Matthew   Ethan
2001 Emily     Madison   Hannah    Ashley    Alexis    Jacob    Michael   Matthew   Joshua    Christopher
```

****12.33** (*Search Web*) Modify Listing 12.18 WebCrawler.java to search for the word (e.g., Computer Programming) starting from a URL (e.g., http://cs.armstrong.edu/liang). Your program prompts the user to enter the word and the starting URL and terminates once the word is found. Display the URL for the page that contains the word.

CHAPTER 13

ABSTRACT CLASSES AND INTERFACES

Objectives

- To design and use abstract classes (§13.2).
- To generalize numeric wrapper classes `BigInteger` and `BigDecimal` using the abstract `Number` class (§13.3).
- To process a calendar using the `Calendar` and `GregorianCalendar` classes (§13.4).
- To specify common behavior for objects using interfaces (§13.5).
- To define interfaces and define classes that implement interfaces (§13.5).
- To define a natural order using the `Comparable` interface (§13.6).
- To make objects cloneable using the `Cloneable` interface (§13.7).
- To explore the similarities and differences among concrete classes, abstract classes, and interfaces (§13.8).
- To design the `Rational` class for processing rational numbers (§13.9).
- To design classes that follow the class-design guidelines (§13.10).

13.1 Introduction

Key Point

A superclass defines common behavior for related subclasses. An interface can be used to define common behavior for classes (including unrelated classes).

problem

interface

You can use the **`java.util.Arrays.sort`** method to sort an array of numbers or strings. Can you apply the same **`sort`** method to sort an array of geometric objects? In order to write such code, you have to know about interfaces. An *interface* is for defining common behavior for classes (including unrelated classes). Before discussing interfaces, we introduce a closely related subject: abstract classes.

13.2 Abstract Classes

Key Point

An abstract class cannot be used to create objects. An abstract class can contain abstract methods that are implemented in concrete subclasses.

abstract class

In the inheritance hierarchy, classes become more specific and concrete *with each new subclass*. If you move from a subclass back up to a superclass, the classes become more general and less specific. Class design should ensure a superclass contains common features of its subclasses. Sometimes, a superclass is so abstract it cannot be used to create any specific instances. Such a class is referred to as an *abstract class*.

VideoNote

Abstract GeometricObject class

abstract method

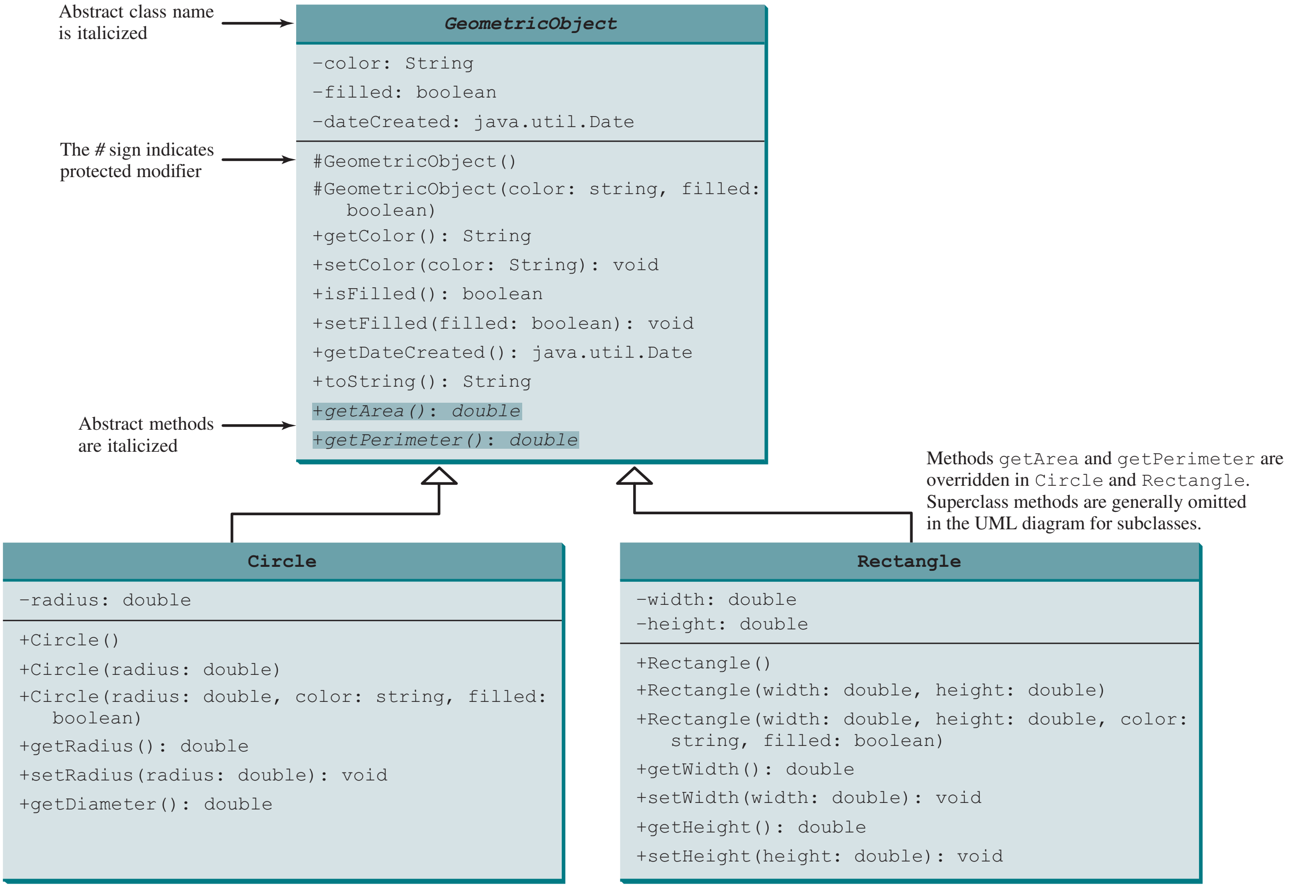
abstract modifier

In Chapter 11, **`GeometricObject`** was defined as the superclass for **`Circle`** and **`Rectangle`**. **`GeometricObject`** models common features of geometric objects. Both **`Circle`** and **`Rectangle`** contain the **`getArea()`** and **`getPerimeter()`** methods for computing the area and perimeter of a circle and a rectangle. Since you can compute areas and perimeters for all geometric objects, it is better to define the **`getArea()`** and **`getPerimeter()`** methods in the **`GeometricObject`** class. However, these methods cannot be implemented in the **`GeometricObject`** class because their implementation depends on the specific type of geometric object. Such methods are referred to as *abstract methods* and are denoted using the **`abstract`** modifier in the method header. After you define the methods in **`GeometricObject`**, it becomes an abstract class. Abstract classes are denoted using the **`abstract`** modifier in the class header. In UML graphic notation, the names of abstract classes and their abstract methods are italicized, as shown in Figure 13.1. Listing 13.1 gives the source code for the new **`GeometricObject`** class.

LISTING 13.1 GeometricObject.java

abstract class

```java
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /** Construct a default geometric object */
  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }

  /** Construct a geometric object with color and filled value */
  protected GeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }

  /** Return color */
  public String getColor() {
    return color;
```


GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
#GeometricObject()
#GeometricObject(color: string, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
+getArea(): double
+getPerimeter(): double
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double, color: string, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void

Abstract class name is italicized

The # sign indicates protected modifier

Abstract methods are italicized

Methods `getArea` and `getPerimeter` are overridden in `Circle` and `Rectangle`. Superclass methods are generally omitted in the UML diagram for subclasses.

FIGURE 13.1 The new **`GeometricObject`** class contains abstract methods.

```
  }

  /** Set a new color */
  public void setColor(String color) {
    this.color = color;
  }

  /** Return filled. Since filled is boolean,
   *  the getter method is named isFilled */
  public boolean isFilled() {
    return filled;
  }

  /** Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }

  /** Get dateCreated */
  public java.util.Date getDateCreated() {
    return dateCreated;
  }
```

abstract method

abstract method

```

  @Override
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }

  /** Abstract method getArea */
  public abstract double getArea();

  /** Abstract method getPerimeter */
  public abstract double getPerimeter();
}
```

Abstract classes are like regular classes, but you cannot create instances of abstract classes using the **`new`** operator. An abstract method is defined without implementation. Its implementation is provided by the subclasses. A class that contains abstract methods must be defined as abstract.

why protected constructor?

The constructor in the abstract class is defined as protected because it is used only by subclasses. When you create an instance of a concrete subclass, its superclass's constructor is invoked to initialize data fields defined in the superclass.

The **`GeometricObject`** abstract class defines the common features (data and methods) for geometric objects and provides appropriate constructors. Because you don't know how to compute areas and perimeters of geometric objects, **`getArea()`** and **`getPerimeter()`** are defined as abstract methods. These methods are implemented in the subclasses. The implementation of **`Circle`** and **`Rectangle`** is the same as in Listings 11.2 and 11.3, except they extend the **`GeometricObject`** class defined in this chapter. You can see the complete code for these two programs at liveexample.pearsoncmg.com/html/Circle.html and liveexample.pearsoncmg.com/html/Rectangle.html, respectively.

implement `Circle`
implement `Rectangle`

LISTING 13.2 `Circle.java`

extends abstract `GeometricObject`

```
public class Circle extends GeometricObject {
  // Same as lines 2-47 in Listing 11.2, so omitted
}
```

LISTING 13.3 `Rectangle.java`

extends abstract `GeometricObject`

```
public class Rectangle extends GeometricObject {
  // Same as lines 2-49 in Listing 11.3, so omitted
}
```

13.2.1 Why Abstract Methods?

You may be wondering what advantage is gained by defining the methods **`getArea()`** and **`getPerimeter()`** as abstract in the **`GeometricObject`** class. The example in Listing 13.4 shows the benefits of defining them in the **`GeometricObject`** class. The program creates two geometric objects, a circle and a rectangle, invokes the **`equalArea`** method to check whether they have equal areas, and invokes the **`displayGeometricObject`** method to display them.

LISTING 13.4 `TestGeometricObject.java`

create a circle
create a rectangle

```
public class TestGeometricObject {
  /** Main method */
  public static void main(String[] args) {
    // Create two geometric objects
    GeometricObject geoObject1 = new Circle(5);
    GeometricObject geoObject2 = new Rectangle(5, 3);
```

```
 7
 8       System.out.println("The two objects have the same area? " +
 9         equalArea(geoObject1, geoObject2));
10
11       // Display circle
12       displayGeometricObject(geoObject1);
13
14       // Display rectangle
15       displayGeometricObject(geoObject2);
16     }
17
18     /** A method for comparing the areas of two geometric objects */
19     public static boolean equalArea(GeometricObject object1,
20         GeometricObject object2) {
21       return object1.getArea() == object2.getArea();
22     }
23
24     /** A method for displaying a geometric object */
25     public static void displayGeometricObject(GeometricObject object) {
26       System.out.println();
27       System.out.println("The area is " + object.getArea());
28       System.out.println("The perimeter is " + object.getPerimeter());
29     }
30   }
```

`equalArea`

`displayGeometricObject`

```
The two objects have the same area? false

The area is 78.53981633974483
The perimeter is 31.41592653589793

The area is 15.0
The perimeter is 16.0
```

The methods **`getArea()`** and **`getPerimeter()`** defined in the **`GeometricObject`** class are overridden in the **`Circle`** class and the **`Rectangle`** class. The statements (lines 5–6)

```
GeometricObject geoObject1 = new Circle(5);
GeometricObject geoObject2 = new Rectangle(5, 3);
```

create a new circle and rectangle and assign them to the variables **`geoObject1`** and **`geoObject2`**. These two variables are of the **`GeometricObject`** type.

When invoking **`equalArea(geoObject1, geoObject2)`** (line 9), the **`getArea()`** method defined in the **`Circle`** class is used for **`object1.getArea()`**, since **`geoObject1`** is a circle, and the **`getArea()`** method defined in the **`Rectangle`** class is used for **`object2.getArea()`**, since **`geoObject2`** is a rectangle.

Similarly, when invoking **`displayGeometricObject(geoObject1)`** (line 12), the methods **`getArea()`** and **`getPerimeter()`** defined in the **`Circle`** class are used, and when invoking **`displayGeometricObject(geoObject2)`** (line 15), the methods **`getArea`** and **`getPerimeter`** defined in the **`Rectangle`** class are used. The JVM dynamically determines which of these methods to invoke at runtime, depending on the actual object that invokes the method.

Note you could not define the **`equalArea`** method for comparing whether two geometric objects have the same area if the **`getArea`** method was not defined in **`GeometricObject`**, since **`object1`** and **`object2`** are of the **`GeometricObject`** type. Now you have seen the benefits of defining the abstract methods in **`GeometricObject`**.

why abstract methods?

13.2.2 Interesting Points about Abstract Classes

The following points about abstract classes are worth noting:

abstract method in abstract class

- An abstract method cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined as abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented. Also note abstract methods are nonstatic.

object cannot be created from abstract class

- An abstract class cannot be instantiated using the **`new`** operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of **`GeometricObject`** are invoked in the **`Circle`** class and the **`Rectangle`** class.

abstract class without abstract method

- A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn't contain any abstract methods. This abstract class is used as a base class for defining subclasses.

concrete method overridden to be abstract

- A subclass can override a method from its superclass to define it as abstract. This is *very unusual*, but it is useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined as abstract.

concrete method overridden to be abstract

- A subclass can be abstract even if its superclass is concrete. For example, the **`Object`** class is concrete, but its subclasses, such as **`GeometricObject`**, may be abstract.

abstract class as type

- You cannot create an instance from an abstract class using the **`new`** operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of the **`GeometricObject`** type, is correct:

  ```
  GeometricObject[] objects = new GeometricObject[10];
  ```

  You can then create an instance of **`GeometricObject`** and assign its reference to the array like this:

  ```
  objects[0] = new Circle();
  ```

Check Point

13.2.1 Which of the following classes defines a legal abstract class?

```
class A {
  abstract void unfinished() {
  }
}
```

(a)

```
public class abstract A {
  abstract void unfinished();
}
```

(b)

```
class A {
  abstract void unfinished();
}
```

(c)

```
abstract class A {
  protected void unfinished();
}
```

(d)

```
abstract class A {
  abstract void unfinished();
}
```

(e)

```
abstract class A {
  abstract int unfinished();
}
```

(f)

13.2.2 The `getArea()` and `getPerimeter()` methods may be removed from the `GeometricObject` class. What are the benefits of defining `getArea()` and `getPerimeter()` as abstract methods in the `GeometricObject` class?

13.2.3 True or false?

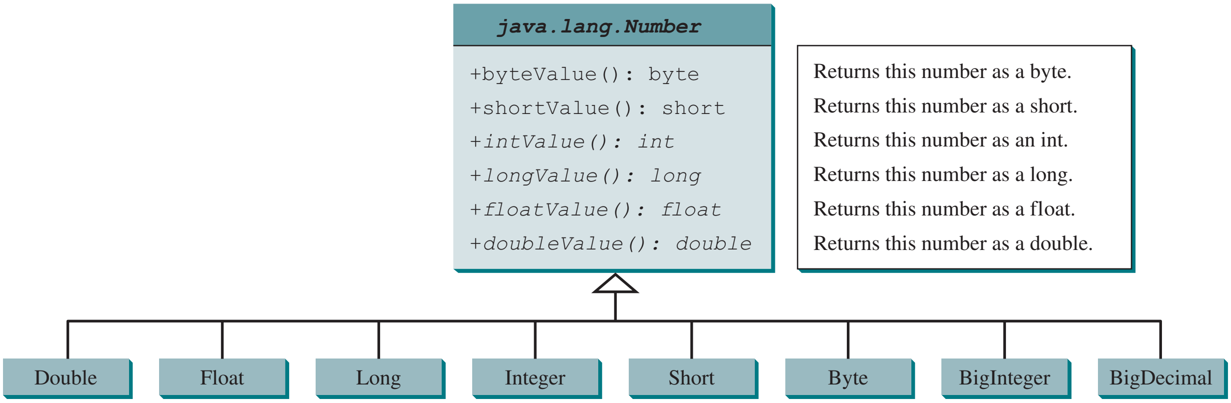
a. An abstract class can be used just like a nonabstract class except that you cannot use the `new` operator to create an instance from the abstract class.
b. An abstract class can be extended.
c. A subclass of a nonabstract superclass cannot be abstract.
d. A subclass cannot override a concrete method in a superclass to define it as abstract.
e. An abstract method must be nonstatic.

13.3 Case Study: The Abstract **Number** Class

Key Point

`Number` is an abstract superclass for numeric wrapper classes `BigInteger` and `BigDecimal`.

Section 10.7 introduced numeric wrapper classes and Section 10.9 introduced the `BigInteger` and `BigDecimal` classes. These classes have common methods `byteValue()`, `shortValue()`, `intValue()`, `longValue()`, `floatValue()`, and `doubleValue()` for returning a `byte`, `short`, `int`, `long`, `float`, and `double` value from an object of these classes. These common methods are actually defined in the `Number` class, which is a superclass for the numeric wrapper classes `BigInteger` and `BigDecimal`, as shown in Figure 13.2.



java.lang.Number
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
Returns this number as a byte.
Returns this number as a short.
Returns this number as an int.
Returns this number as a long.
Returns this number as a float.
Returns this number as a double.
Double
Float
Long
Integer
Short
Byte
BigInteger
BigDecimal

FIGURE 13.2 The `Number` class is an abstract superclass for `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `BigInteger`, and `BigDecimal`.

Since the `intValue()`, `longValue()`, `floatValue()`, and `doubleValue()` methods cannot be implemented in the `Number` class, they are defined as abstract methods in the `Number` class. The `Number` class is therefore an abstract class. The `byteValue()` and `shortValue()` methods are implemented from the `intValue()` method as follows:

```
public byte byteValue() {
  return (byte)intValue();
}

public short shortValue() {
  return (short)intValue();
}
```

With **`Number`** defined as the superclass for the numeric classes, we can define methods to perform common operations for numbers. Listing 13.5 gives a program that finds the largest number in a list of **`Number`** objects.

LISTING 13.5 LargestNumber.java

create an array list

add number to list

invoke `getLargestNumber`

`doubleValue`

```
 1 import java.util.ArrayList;
 2 import java.math.*;
 3
 4 public class LargestNumber {
 5   public static void main(String[] args) {
 6     ArrayList<Number> list = new ArrayList<>();
 7     list.add(45); // Add an integer
 8     list.add(3445.53); // Add a double
 9     // Add a BigInteger
10     list.add(new BigInteger("3432323234344343101"));
11     // Add a BigDecimal
12     list.add(new BigDecimal("2.0909090989091343433344343"));
13
14     System.out.println("The largest number is " +
15       getLargestNumber(list));
16   }
17
18   public static Number getLargestNumber(ArrayList<Number> list) {
19     if (list == null || list.size() == 0)
20       return null;
21
22     Number number = list.get(0);
23     for (int i = 1; i < list.size(); i++)
24       if (number.doubleValue() < list.get(i).doubleValue())
25         number = list.get(i);
26
27     return number;
28   }
29 }
```

```
The largest number is 3432323234344343101
```

The program creates an **`ArrayList`** of **`Number`** objects (line 6). It adds an **`Integer`** object, a **`Double`** object, a **`BigInteger`** object, and a **`BigDecimal`** object to the list (lines 7–12). Note **`45`** is automatically converted into an **`Integer`** object and added to the list in line 7, and **`3445.53`** is automatically converted into a **`Double`** object and added to the list in line 8 using autoboxing.

Invoking the **`getLargestNumber`** method returns the largest number in the list (line 15). The **`getLargestNumber`** method returns **`null`** if the list is **`null`** or the list size is **`0`** (lines 19 and 20). To find the largest number in the list, the numbers are compared by invoking their **`doubleValue()`** method (line 24). The **`doubleValue()`** method is defined in the **`Number`** class and implemented in the concrete subclass of **`Number`**. If a number is an **`Integer`** object, the **`Integer`**'s **`doubleValue()`** is invoked. If a number is a **`BigDecimal`** object, the **`BigDecimal`**'s **`doubleValue()`** is invoked.

If the **`doubleValue()`** method was not defined in the **`Number`** class, you will not be able to find the largest number among different types of numbers using the **`Number`** class.

Check Point

13.3.1 Why do the following two lines of code compile but cause a runtime error?

```
Number numberRef = Integer.valueOf(0);
Double doubleRef = (Double)numberRef;
```

13.3.2 Why do the following two lines of code compile but cause a runtime error?

```
Number[] numberArray = Integer[2];
numberArray[0] = Double.valueOf(1.5);
```

13.3.3 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    Number x = 3;
    System.out.println(x.intValue());
    System.out.println(x.doubleValue());
  }
}
```

13.3.4 What is wrong in the following code? (Note the `compareTo` method for the `Integer` and `Double` classes was introduced in Section 10.7.)

```
public class Test {
  public static void main(String[] args) {
    Number x = Integer.valueOf(3);
    System.out.println(x.intValue());
    System.out.println(x.compareTo(4));
  }
}
```

13.3.5 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    Number x = Integer.valueOf(3);
    System.out.println(x.intValue());
    System.out.println((Integer)x.compareTo(4));
  }
}
```

13.4 Case Study: `Calendar` and `GregorianCalendar`

`GregorianCalendar` is a concrete subclass of the abstract class `Calendar`.

Key Point

VideoNote
Calendar and GregorianCalendar classes

An instance of `java.util.Date` represents a specific instant in time with millisecond precision. `java.util.Calendar` is an abstract base class for extracting detailed calendar information, such as the year, month, date, hour, minute, and second. Subclasses of `Calendar` can implement specific calendar systems, such as the Gregorian calendar, the lunar calendar, and the Jewish calendar. Currently, `java.util.GregorianCalendar` for the Gregorian calendar is supported in Java, as shown in Figure 13.3. The `add` method is abstract in the `Calendar` class because its implementation is dependent on a concrete calendar system.

abstract add method

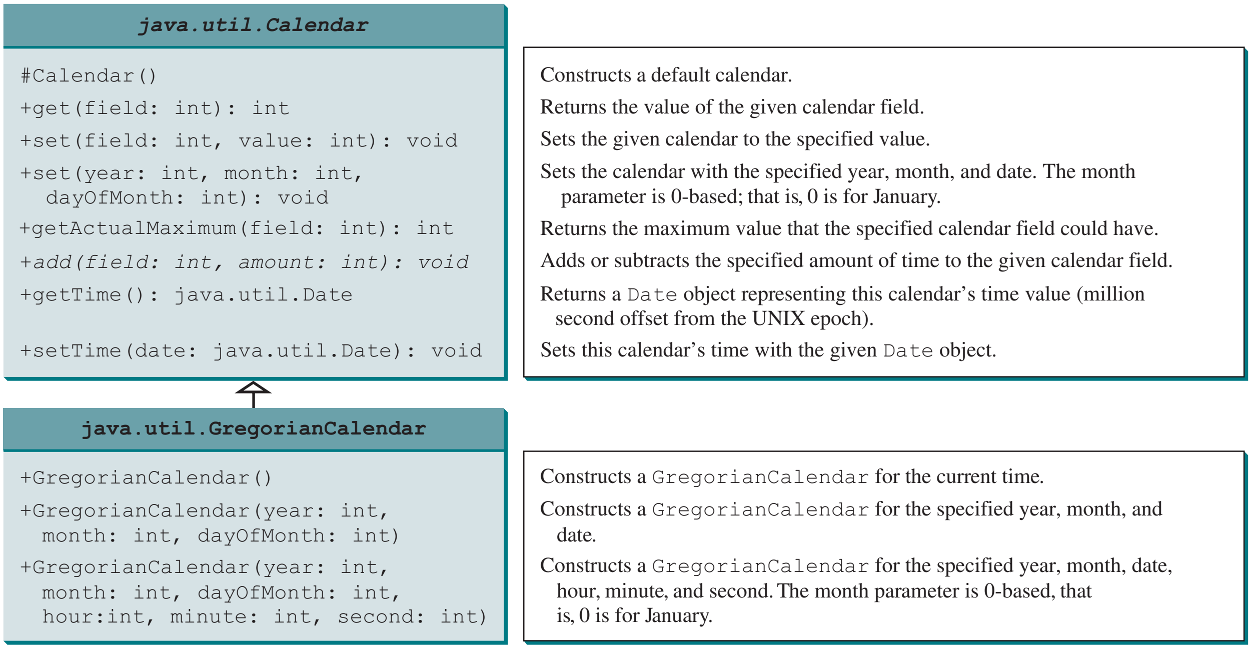
construct calendar

You can use `new GregorianCalendar()` to construct a default `GregorianCalendar` with the current time and `new GregorianCalendar(year, month, date)` to construct a `GregorianCalendar` with the specified `year`, `month`, and `date`. The `month` parameter is `0`-based—that is, `0` is for January.

`get(field)`

The `get(int field)` method defined in the `Calendar` class is useful for extracting the date and time information from a `Calendar` object. The fields are defined as constants, as shown in Table 13.1.

Listing 13.6 gives an example that displays the date and time information for the current time.



java.util.Calendar	
`#Calendar()`	Constructs a default calendar.
`+get(field: int): int`	Returns the value of the given calendar field.
`+set(field: int, value: int): void`	Sets the given calendar to the specified value.
`+set(year: int, month: int, dayOfMonth: int): void`	Sets the calendar with the specified year, month, and date. The month parameter is 0-based; that is, 0 is for January.
`+getActualMaximum(field: int): int`	Returns the maximum value that the specified calendar field could have.
`+add(field: int, amount: int): void`	Adds or subtracts the specified amount of time to the given calendar field.
`+getTime(): java.util.Date`	Returns a `Date` object representing this calendar's time value (million second offset from the UNIX epoch).
`+setTime(date: java.util.Date): void`	Sets this calendar's time with the given `Date` object.

java.util.GregorianCalendar	
`+GregorianCalendar()`	Constructs a `GregorianCalendar` for the current time.
`+GregorianCalendar(year: int, month: int, dayOfMonth: int)`	Constructs a `GregorianCalendar` for the specified year, month, and date.
`+GregorianCalendar(year: int, month: int, dayOfMonth: int, hour:int, minute: int, second: int)`	Constructs a `GregorianCalendar` for the specified year, month, date, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.

FIGURE 13.3 The abstract `Calendar` class defines common features of various calendars.

TABLE 13.1 Field Constants in the `Calendar` Class

Constant	*Description*
`YEAR`	The year of the calendar.
`MONTH`	The month of the calendar, with 0 for January.
`DATE`	The day of the calendar.
`HOUR`	The hour of the calendar (12-hour notation).
`HOUR_OF_DAY`	The hour of the calendar (24-hour notation).
`MINUTE`	The minute of the calendar.
`SECOND`	The second of the calendar.
`DAY_OF_WEEK`	The day number within the week, with 1 for Sunday.
`DAY_OF_MONTH`	Same as DATE.
`DAY_OF_YEAR`	The day number in the year, with 1 for the first day of the year.
`WEEK_OF_MONTH`	The week number within the month, with 1 for the first week.
`WEEK_OF_YEAR`	The week number within the year, with 1 for the first week.
`AM_PM`	Indicator for AM or PM (0 for AM and 1 for PM).

LISTING 13.6 TestCalendar.java

calendar for current time

extract fields in calendar

```java
import java.util.*;

public class TestCalendar {
  public static void main(String[] args) {
    // Construct a Gregorian calendar for the current date and time
    Calendar calendar = new GregorianCalendar();
    System.out.println("Current time is " + new Date());
    System.out.println("YEAR: " + calendar.get(Calendar.YEAR));
    System.out.println("MONTH: " + calendar.get(Calendar.MONTH));
    System.out.println("DATE: " + calendar.get(Calendar.DATE));
    System.out.println("HOUR: " + calendar.get(Calendar.HOUR));
    System.out.println("HOUR_OF_DAY: " +
      calendar.get(Calendar.HOUR_OF_DAY));
    System.out.println("MINUTE: " + calendar.get(Calendar.MINUTE));
    System.out.println("SECOND: " + calendar.get(Calendar.SECOND));
    System.out.println("DAY_OF_WEEK: " +
      calendar.get(Calendar.DAY_OF_WEEK));
    System.out.println("DAY_OF_MONTH: " +
      calendar.get(Calendar.DAY_OF_MONTH));
    System.out.println("DAY_OF_YEAR: " +
      calendar.get(Calendar.DAY_OF_YEAR));
    System.out.println("WEEK_OF_MONTH: " +
      calendar.get(Calendar.WEEK_OF_MONTH));
    System.out.println("WEEK_OF_YEAR: " +
      calendar.get(Calendar.WEEK_OF_YEAR));
    System.out.println("AM_PM: " + calendar.get(Calendar.AM_PM));

    // Construct a calendar for December 25, 1997
    Calendar calendar1 = new GregorianCalendar(1997, 11, 25);
    String[] dayNameOfWeek = {"Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday"};
    System.out.println("December 25, 1997 is a " +
      dayNameOfWeek[calendar1.get(Calendar.DAY_OF_WEEK) - 1]);
  }
}
```

create a calendar

```
Current time is Tue Sep 22 12:55:56 EDT 2015
YEAR: 2015
MONTH: 8
DATE: 22
HOUR: 0
HOUR_OF_DAY: 12
MINUTE: 55
SECOND: 56
DAY_OF_WEEK: 3
DAY_OF_MONTH: 22
DAY_OF_YEAR: 265
WEEK_OF_MONTH: 4
WEEK_OF_YEAR: 39
AM_PM: 1
December 25, 1997 is a Thursday
```

`set(field, value)`

The `set(int field, value)` method defined in the `Calendar` class can be used to set a field. For example, you can use `calendar.set(Calendar.DAY_OF_MONTH, 1)` to set the `calendar` to the first day of the month.

`add(field, amount)`

The `add(field, value)` method adds the specified amount to a given field. For example, `add(Calendar.DAY_OF_MONTH, 5)` adds five days to the current time of the calendar. `add(Calendar.DAY_OF_MONTH, -5)` subtracts five days from the current time of the calendar.

`getActualMaximum(field)`

To obtain the number of days in a month, use `calendar.getActualMaximum(Calendar.DAY_OF_MONTH)`. For example, if the `calendar` was for March, this method would return `31`.

`setTime(date)`
`getTime()`

You can set a time represented in a `Date` object for the `calendar` by invoking `calendar.setTime(date)` and retrieve the time by invoking `calendar.getTime()`.

Check Point

13.4.1 Can you create a `Calendar` object using the `Calendar` class?

13.4.2 Which method in the `Calendar` class is abstract?

13.4.3 How do you create a `Calendar` object for the current time?

13.4.4 For a `Calendar` object `c`, how do you get its year, month, date, hour, minute, and second?

13.5 Interfaces

Key Point

An interface is a class-like construct for defining common operations for objects.

VideoNote
The concept of interface

In many ways an interface is similar to an abstract class, but its intent is to specify common behavior for objects of related classes or unrelated classes. For example, using appropriate interfaces, you can specify that the objects are comparable, edible, and/or cloneable.

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
modifier interface InterfaceName {
  /** Constant declarations */
  /** Abstract method signatures */
}
```

Here is an example of an interface:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. You can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a reference variable, as the result of casting, and so on. As with an abstract class, you cannot create an instance from an interface using the `new` operator.

interface inheritance

You can use the `Edible` interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the `implements` keyword. For example, the classes `Chicken` and `Fruit` in Listing 13.7 (lines 30 and 49) implement the `Edible` interface. The relationship between the class and the interface is known as *interface inheritance*. Since interface inheritance and class inheritance are essentially the same, we will simply refer to both as *inheritance*.

LISTING 13.7 TestEdible.java

```
 1  public class TestEdible {
 2    public static void main(String[] args) {
 3      Object[] objects = {new Tiger(), new Chicken(), new Apple()};
 4      for (int i = 0; i < objects.length; i++) {
 5        if (objects[i] instanceof Edible)
 6          System.out.println(((Edible)objects[i]).howToEat());
 7
 8        if (objects[i] instanceof Animal) {
 9          System.out.println(((Animal)objects[i]).sound());
10        }
11      }
12    }
13  }
```

Animal class

implements Edible

howToEat()

Tiger class

implements Edible

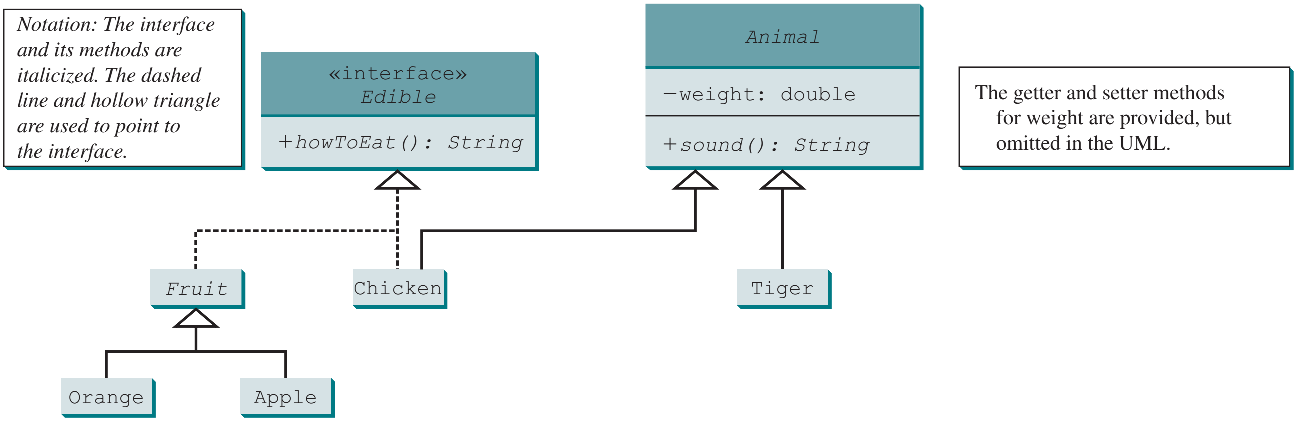
Apple class

Orange class

```

abstract class Animal {
  private double weight;

  public double getWeight() {
    return weight;
  }

  public void setWeight(double weight) {
    this.weight = weight;
  }

  /** Return animal sound */
  public abstract String sound();
}

class Chicken extends Animal implements Edible {
  @Override
  public String howToEat() {
    return "Chicken: Fry it";
  }

  @Override
  public String sound() {
    return "Chicken: cock-a-doodle-doo";
  }
}

class Tiger extends Animal {
  @Override
  public String sound() {
    return "Tiger: RROOAARR";
  }
}

abstract class Fruit implements Edible {
  // Data fields, constructors, and methods omitted here
}

class Apple extends Fruit {
  @Override
  public String howToEat() {
    return "Apple: Make apple cider";
  }
}

class Orange extends Fruit {
  @Override
  public String howToEat() {
    return "Orange: Make orange juice";
  }
}
```

```
Tiger: RROOAARR
Chicken: Fry it
Chicken: cock-a-doodle-doo
Apple: Make apple cider
```

This example uses several classes and interfaces. Their inheritance relationship is shown in Figure 13.4.



Notation: The interface and its methods are italicized. The dashed line and hollow triangle are used to point to the interface.

The getter and setter methods for weight are provided, but omitted in the UML.

FIGURE 13.4 `Edible` is a supertype for `Chicken` and `Fruit`. `Animal` is a supertype for `Chicken` and `Tiger`. `Fruit` is a supertype for `Orange` and `Apple`.

The `Animal` class defines the `weight` property with its getter and setter methods (lines 16–24) and the `sound` method (line 27). The `sound` method is an abstract method and will be implemented by a concrete `animal` class.

The `Chicken` class implements `Edible` to specify that chickens are edible. When a class implements an interface, it implements all the methods defined in the interface. The `Chicken` class implements the `howToEat` method (lines 32–34). `Chicken` also extends `Animal` to implement the `sound` method (lines 37–39).

The `Fruit` class implements `Edible`. Since it does not implement the `howToEat` method, `Fruit` must be defined as `abstract` (line 49). The concrete subclasses of `Fruit` must implement the `howToEat` method. The `Apple` and `Orange` classes implement the `howToEat` method (lines 55 and 62).

The `main` method creates an array with three objects for `Tiger`, `Chicken`, and `Apple` (line 3) and invokes the `howToEat` method if the element is edible (line 6), and the `sound` method if the element is an animal (line 9).

common behavior

In essence, the `Edible` interface defines common behavior for edible objects. All edible objects have the `howToEat` method.

omit modifiers

Note

The modifiers ***`public static final`*** on data fields and the modifiers ***`public abstract`*** on methods can be omitted in an interface. Therefore, the following interface definitions are equivalent:

```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T {
  int K = 1;

  void p();
}
```

Although the `public` modifier may be omitted for a method defined in the interface, the method must be defined `public` when it is implemented in a subclass.

default methods

Note

Java 8 introduced default interface methods using the keyword `default`. A default method provides a default implementation for the method in the interface. A class that implements the interface may simply use the default implementation for the method or override the method with a new implementation. This feature enables you to add a new method to an existing interface with a default implementation without having to rewrite the code for the existing classes that implement this interface.

Java 8 also permits public static methods in an interface. A public static method in an interface can be used just like a public static method in a class.

In Java 9, you can also use private methods in an interface. These methods are used for implementing the default methods and public static methods. Here is an example of defining default methods, static methods, and private methods in an interface:

```
public interface Java89Interface {
  /** default method in Java 8*/
  public default void doSomething() {
    System.out.println("Do something");
  }

  /** static method in Java 8*/
  public static int getAValue() {
    return 0;
  }

  /** private static method Java 9 */
  private static int getAStaticValue() {
    return 0;
  }

  /** private instance method Java 9 */
  private void performPrivateAction() {
  }
}
```

Check Point

13.5.1 Suppose **A** is an interface. Can you create an instance using `new A()`?

13.5.2 Suppose **A** is an interface. Can you declare a reference variable **x** with type **A** like this?

```
A x;
```

13.5.3 Which of the following is a correct interface?

```
interface A {
  void print() { }
}
```

(a)

```
abstract interface A {
  abstract void print() { }
}
```

(b)

```
abstract interface A {
  print();
}
```

(c)

```
interface A {
  void print();
}
```

(d)

```
interface A {
  default void print() {
  }
}
```

(e)

```
interface A {
  static int get() {
    return 0;
  }
}
```

(f)

13.5.4 Show the error in the following code:

```
interface A {
  void m1();
}

class B implements A {
  void m1() {
    System.out.println("m1");
  }
}
```

13.6 The `Comparable` Interface

Key Point

*The **`Comparable`** interface defines the **`compareTo`** method for comparing objects.*

Suppose you want to design a generic method to find the larger of two objects of the same type, such as two students, two dates, two circles, two rectangles, or two squares. In order to accomplish this, the two objects must be comparable, so the common behavior for the objects must be comparable. Java provides the **`Comparable`** interface for this purpose. The interface is defined as follows:

java.lang.Comparable

```
// Interface for comparing objects, defined in java.lang
package java.lang;

public interface Comparable<E> {
  public int compareTo(E o);
}
```

The **`compareTo`** method determines the order of this object with the specified object **`o`** and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than **`o`**.

The **`Comparable`** interface is a generic interface. The generic type **`E`** is replaced by a concrete type when implementing this interface. Many classes in the Java library implement **`Comparable`** to define a natural order for objects. The classes **`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**, **`Character`**, **`BigInteger`**, **`BigDecimal`**, **`Calendar`**, **`String`**, and **`Date`** all implement the **`Comparable`** interface. For example, the **`Integer`**, **`BigInteger`**, **`String`**, and **`Date`** classes are defined as follows in the Java API:

```
public final class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```
public class BigInteger extends Number
    implements Comparable<Biginteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

```
public final class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

Thus, numbers are comparable, strings are comparable, and so are dates. You can use the **`compareTo`** method to compare two numbers, two strings, and two dates. For example, the following code:

```
System.out.println(Integer.valueOf(3).compareTo(5));
System.out.println("ABC".compareTo("ABC"));
java.util.Date date1 = new java.util.Date(2013, 1, 1);
java.util.Date date2 = new java.util.Date(2012, 1, 1);
System.out.println(date1.compareTo(date2));
```

displays

```
-1
 0
 1
```

Line 1 displays a negative value since `3` is less than `5`. Line 2 displays zero since `ABC` is equal to `ABC`. Line 5 displays a positive value since `date1` is greater than `date2`.

Let `n` be an `Integer` object, `s` be a `String` object, and `d` be a `Date` object. All the following expressions are `true`:

```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof Date
d instanceof Object
d instanceof Comparable
```

Since all `Comparable` objects have the `compareTo` method, the `java.util.Arrays.sort(Object[])` method in the Java API uses the `compareTo` method to compare and sorts the objects in an array, provided the objects are instances of the `Comparable` interface. Listing 13.8 gives an example of sorting an array of strings and an array of `BigInteger` objects.

LISTING 13.8 SortComparableObjects.java

```
 1  import java.math.*;
 2
 3  public class SortComparableObjects {
 4    public static void main(String[] args) {
 5      String[] cities = {"Savannah", "Boston", "Atlanta", "Tampa"};
 6      java.util.Arrays.sort(cities);
 7      for (String city: cities)
 8        System.out.print(city + " ");
 9      System.out.println();
10
11      BigInteger[] hugeNumbers = {new BigInteger("2323231092923992"),
12        new BigInteger("432232323239292"),
13        new BigInteger("54623239292")};
14      java.util.Arrays.sort(hugeNumbers);
15      for (BigInteger number: hugeNumbers)
16        System.out.print(number + " ");
17    }
18  }
```

create an array

sort the array

create an array

sort the array

```
Atlanta Boston Savannah Tampa
54623239292 432232323239292 2323231092923992
```

The program creates an array of strings (line 5) and invokes the `sort` method to sort the strings (line 6). The program creates an array of `BigInteger` objects (lines 11–13) and invokes the `sort` method to sort the `BigInteger` objects (line 14).

You cannot use the `sort` method to sort an array of `Rectangle` objects because `Rectangle` does not implement `Comparable`. However, you can define a new rectangle class that implements `Comparable`. The instances of this new class are comparable. Let this new class be named `ComparableRectangle`, as shown in Listing 13.9.

LISTING 13.9 ComparableRectangle.java

```
public class ComparableRectangle extends Rectangle
    implements Comparable<ComparableRectangle> {
  /** Construct a ComparableRectangle with specified properties */
  public ComparableRectangle(double width, double height) {
    super(width, height);
  }

  @Override // Implement the compareTo method defined in Comparable
```

implements `Comparable`

implement `compareTo`

implement `toString`

```java
  public int compareTo(ComparableRectangle o) {
    if (getArea() > o.getArea())
      return 1;
    else if (getArea() < o.getArea())
      return -1;
    else
      return 0;
  }

  @Override // Implement the toString method in GeometricObject
  public String toString() {
    return "Width: " + getWidth() + " Height: " + getHeight() +
      "Area: " + getArea();
  }
}
```

`ComparableRectangle` extends **`Rectangle`** and implements **`Comparable`**, as shown in Figure 13.5. The keyword **`implements`** indicates that **`ComparableRectangle`** inherits all the constants from the **`Comparable`** interface and implements the methods in the interface. The **`compareTo`** method compares the areas of two rectangles. An instance of **`ComparableRectangle`** is also an instance of **`Rectangle`**, **`GeometricObject`**, **`Object`**, and **`Comparable`**.

Notation:
The interface name and the method names are italicized. The dashed line and hollow triangle are used to point to the interface.

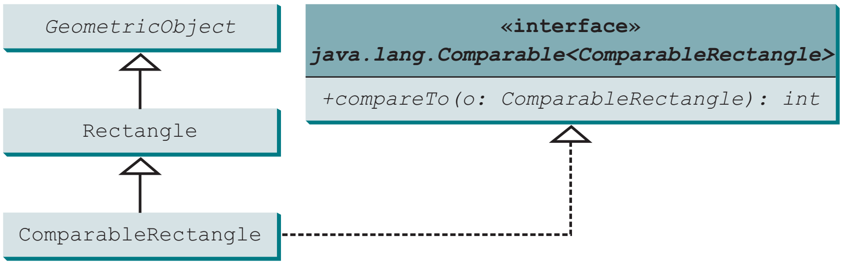

GeometricObject
Rectangle
ComparableRectangle
«interface»
java.lang.Comparable<ComparableRectangle>
+compareTo(o: ComparableRectangle): int

FIGURE 13.5 **`ComparableRectangle`** extends **`Rectangle`** and implements **`Comparable`**.

You can now use the **`sort`** method to sort an array of **`ComparableRectangle`** objects, as in Listing 13.10.

LISTING 13.10 SortRectangles.java

create an array

sort the array

```java
public class SortRectangles {
  public static void main(String[] args) {
    ComparableRectangle[] rectangles = {
      new ComparableRectangle(3.4, 5.4),
      new ComparableRectangle(13.24, 55.4),
      new ComparableRectangle(7.4, 35.4),
      new ComparableRectangle(1.4, 25.4)};
    java.util.Arrays.sort(rectangles);
    for (Rectangle rectangle: rectangles) {
      System.out.print(rectangle + " ");
      System.out.println();
    }
  }
}
```

```
Width: 3.4 Height: 5.4 Area: 18.36
Width: 1.4 Height: 25.4 Area: 35.559999999999995
Width: 7.4 Height: 35.4 Area: 261.96
Width: 13.24 Height: 55.4 Area: 733.496
```

benefits of interface

An interface provides another form of generic programming. It would be difficult to use a generic `sort` method to sort the objects without using an interface in this example, because multiple inheritance would be necessary to inherit `Comparable` and another class, such as `Rectangle`, at the same time.

The `Object` class contains the `equals` method, which is intended for the subclasses of the `Object` class to override in order to compare whether the contents of the objects are the same. Suppose the `Object` class contains the `compareTo` method, as defined in the `Comparable` interface; the `sort` method can be used to compare a list of *any* objects. Whether a `compareTo` method should be included in the `Object` class is debatable. Since the `compareTo` method is not defined in the `Object` class, the `Comparable` interface is defined in Java to enable objects to be compared if they are instances of the `Comparable` interface. `compareTo` should be consistent with `equals`. That is, for two objects `o1` and `o2`, `o1.compareTo(o2) == 0` if and only if `o1.equals(o2)` is `true`. Therefore, you should also override the `equals` method in the `ComparableRectangle` class to return `true` if two rectangles have the same area.

Check Point

13.6.1 True or false? If a class implements `Comparable`, the object of the class can invoke the `compareTo` method.

13.6.2 Which of the following is the correct method header for the `compareTo` method in the `String` class?

```
public int compareTo(String o)
public int compareTo(Object o)
```

13.6.3 Can the following code be compiled? Why?

```
Integer n1 = 3;
Object n2 = 4;
System.out.println(n1.compareTo(n2));
```

13.6.4 You can define the `compareTo` method in a class without implementing the `Comparable` interface. What are the benefits of implementing the `Comparable` interface?

13.6.5 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    Person[] persons = {new Person(3), new Person(4), new Person(1)};
    java.util.Arrays.sort(persons);
  }
}

class Person {
  private int id;

  Person(int id) {
    this.id = id;
  }
}
```

13.6.6 Simplify the code in lines 10–15 in Listing 13.9 using one line of code. Also override the `equals` method in this class.

13.6.7 Listing 13.5 has an error. If you add `list.add(new BigInteger("3432323234344343102"));` in line 11, you will see the result is incorrect. This is due to the fact that a double value can have up to 17 significant digits. When invoking `doubleValue()` on a `BigInteger` object in line 24, precision is lost. Fix the error by converting the numbers into `BigDecimal`, and compare them using the `compareTo` method in line 24.

13.7 The `Cloneable` Interface

Key Point

The `Cloneable` interface specifies that an object can be cloned.

Often, it is desirable to create a copy of an object. To do this, you need to use the `clone` method and understand the `Cloneable` interface.

An interface contains constants and abstract methods, but the `Cloneable` interface is a special case. The `Cloneable` interface in the `java.lang` package is defined as follows:

java.lang.Cloneable

```
package java.lang;

public interface Cloneable {
}
```

marker interface

This interface is empty. An interface with an empty body is referred to as a *marker interface*. A marker interface is used to denote that a class possesses certain desirable properties. A class that implements the `Cloneable` interface is marked cloneable, and its objects can be cloned using the `clone()` method defined in the `Object` class.

Many classes in the Java library (e.g., `Date`, `Calendar` and `ArrayList`) implement `Cloneable`. Thus, the instances of these classes can be cloned. For example, the following code:

```
1  Calendar calendar = new GregorianCalendar(2013, 2, 1);
2  Calendar calendar1 = calendar;
3  Calendar calendar2 = (Calendar)calendar.clone();
4  System.out.println("calendar == calendar1 is " +
5    (calendar == calendar1));
6  System.out.println("calendar == calendar2 is " +
7    (calendar == calendar2));
8  System.out.println("calendar.equals(calendar2) is " +
9    calendar.equals(calendar2));
```

displays

```
calendar == calendar1 is true
calendar == calendar2 is false
calendar.equals(calendar2) is true
```

In the preceding code, line 2 copies the reference of `calendar` to `calendar1`, so `calendar` and `calendar1` point to the same `Calendar` object. Line 3 creates a new object that is the clone of `calendar` and assigns the new object's reference to `calendar2`. `calendar2` and `calendar` are different objects with the same contents.

The following code:

```
ArrayList<Double> list1 = new ArrayList<>();
list1.add(1.5);
list1.add(2.5);
list1.add(3.5);
ArrayList<Double> list2 = (ArrayList<Double>)list1.clone();
ArrayList<Double> list3 = list1;
list2.add(4.5);
list3.remove(1.5);
System.out.println("list1 is " + list1);
System.out.println("list2 is " + list2);
System.out.println("list3 is " + list3);
```

displays

```
list1 is [2.5, 3.5]
list2 is [1.5, 2.5, 3.5, 4.5]
list3 is [2.5, 3.5]
```

In the preceding code, line 5 creates a new object that is the clone of `list1` and assigns the new object's reference to `list2`. `list2` and `list1` are different objects with the same contents. Line 6 copies the reference of `list1` to `list3`, so `list1` and `list3` point to the same `ArrayList` object. Line 7 adds `4.5` into `list2`. Line 8 removes `1.5` from `list3`. Since `list1` and `list3` point to the same `ArrayList`, lines 9 and 11 display the same content.

`clone` arrays

You can clone an array using the `clone` method. For example, the following code:

```
int[] list1 = {1, 2};
int[] list2 = list1.clone();
list1[0] = 7;
list2[1] = 8;
System.out.println("list1 is " + list1[0] + ", " + list1[1]);
System.out.println("list2 is " + list2[0] + ", " + list2[1]);
```

displays

```
list1 is 7, 2
list2 is 1, 8
```

Note the return type of the `clone()` method for an array is the same as the type of the array. For example, the return type for `list1.clone()` is `int[]` since `list1` is of the type `int[]`.

how to implement `Cloneable`

To define a custom class that implements the `Cloneable` interface, the class must override the `clone()` method in the `Object` class. Listing 13.11 defines a class named `House` that implements `Cloneable` and `Comparable`.

LISTING 13.11 House.java

```
public class House implements Cloneable, Comparable<House> {
  private int id;
  private double area;
  private java.util.Date whenBuilt;

  public House(int id, double area) {
    this.id = id;
    this.area = area;
    whenBuilt = new java.util.Date();
  }

  public int getId() {
    return id;
  }

  public double getArea() {
    return area;
  }

  public java.util.Date getWhenBuilt() {
    return whenBuilt;
  }
```

This exception is thrown if `House` does not implement `Cloneable`

```
23
24    @Override /** Override the protected clone method defined in
25      the Object class, and strengthen its accessibility */
26    public Object clone() {
27      try {
28        return super.clone();
29      }
30      catch (CloneNotSupportedException ex) {
31        return null;
32      }
33    }
34
35    @Override // Implement the compareTo method defined in Comparable
36    public int compareTo(House o) {
37      if (area > o.area)
38        return 1;
39      else if (area < o.area)
40        return -1;
41      else
42        return 0;
43    }
44  }
```

The `House` class implements the `clone` method (lines 26–33) defined in the `Object` class. The header for the `clone` method defined in the `Object` class is

```
protected native Object clone() throws CloneNotSupportedException;
```

`CloneNotSupportedException`

The keyword `native` indicates that this method is not written in Java, but is implemented in the JVM for the native platform. The keyword `protected` restricts the method to be accessed in the same package or in a subclass. For this reason, the `House` class must override the method and change the visibility modifier to `public` so the method can be used in any package. Since the `clone` method implemented for the native platform in the `Object` class performs the task of cloning objects, the `clone` method in the `House` class simply invokes `super.clone()`. The `clone` method defined in the `Object` class throws `CloneNotSupportedException` if the object is not a type of `Cloneable`. Since we catch the exception in the method (lines 30–32), there is no need to declare it in the `clone()` method header.

The `House` class implements the `compareTo` method (lines 36–43) defined in the `Comparable` interface. The method compares the areas of two houses.

You can now create an object of the `House` class and create an identical copy from it, as follows:

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

shallow copy

deep copy

`house1` and `house2` are two different objects with identical contents. The `clone` method in the `Object` class copies each field from the original object to the target object. If the field is of a primitive type, its value is copied. For example, the value of `area` (`double` type) is copied from `house1` to `house2`. If the field is of an object, the reference of the field is copied. For example, the field `whenBuilt` is of the `Date` class, so its reference is copied into `house2`, as shown in Figure 13.6a. Therefore, `house1.whenBuilt == house2.whenBuilt` is true, although `house1 == house2` is false. This is referred to as a *shallow copy* rather than a *deep copy*, meaning if the field is of an object type, the object's reference is copied rather than its contents.

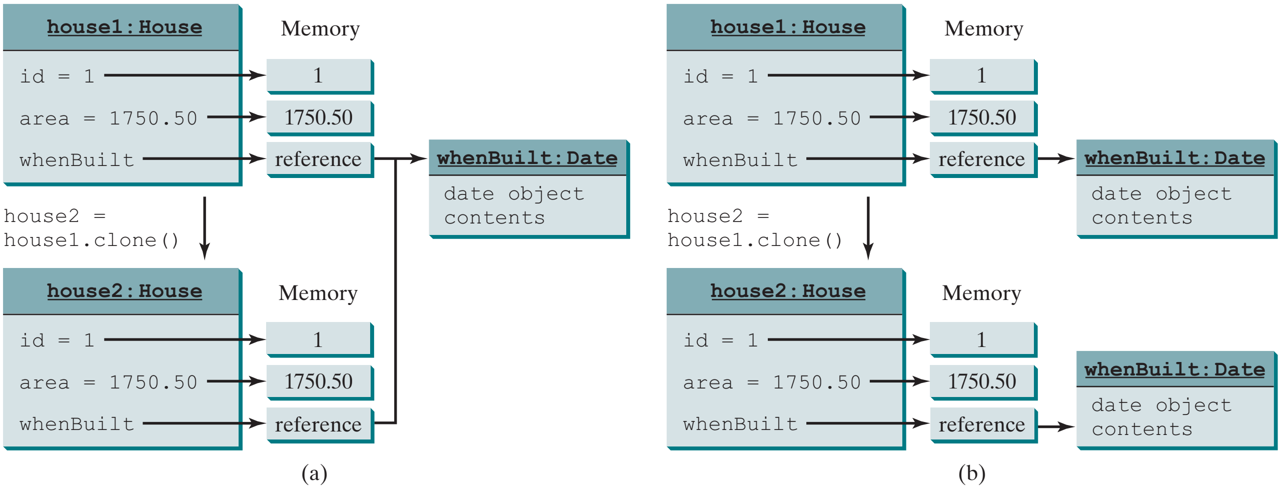

house1: House
id = 1
area = 1750.50
whenBuilt
Memory
1
1750.50
reference
whenBuilt: Date
date object contents
house2 = house1.clone()
house2: House
id = 1
area = 1750.50
whenBuilt
Memory
1
1750.50
reference
(a)
house1: House
id = 1
area = 1750.50
whenBuilt
Memory
1
1750.50
reference
whenBuilt: Date
date object contents
house2 = house1.clone()
house2: House
id = 1
area = 1750.50
whenBuilt
Memory
1
1750.50
reference
whenBuilt: Date
date object contents
(b)

FIGURE 13.6 (a) The default `clone` method performs a shallow copy. (b) The custom `clone` method performs a deep copy.

deep copy

To perform a deep copy for a `House` object, replace the `clone()` method in lines 26–33 with the following code: (For the complete code, see liveexample.pearsoncmg.com/text/House.txt.)

```
public Object clone() throws CloneNotSupportedException {
  // Perform a shallow copy
  House houseClone = (House)super.clone();
  // Deep copy on whenBuilt
  houseClone.whenBuilt = (java.util.Date)(whenBuilt.clone());
  return houseClone;
}
```

or

```
public Object clone() {
  try {
    // Perform a shallow copy
    House houseClone = (House)super.clone();
    // Deep copy on whenBuilt
    houseClone.whenBuilt = (java.util.Date)(whenBuilt.clone());
    return houseClone;
  }
  catch (CloneNotSupportedException ex) {
    return null;
  }
}
```

Now, if you clone a `House` object in the following code:

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

`house1.whenBuilt` == `house2.whenBuilt` will be `false`. `house1` and `house2` contain two different `Date` objects, as shown in Figure 13.6b.

Several questions arise from the `clone` method and `Cloneable` interface.

First, why is the `clone` method in the `Object` class defined protected, not public? Not every object can be cloned. The designer of Java purposely forces the subclasses to override it if an object of the subclass is cloneable.

Second, why is the `clone` method not defined in the `Cloneable` interface? Java provides a native method that performs a shallow copy to clone an object. Since a method in an interface is abstract, this native method cannot be implemented in the interface. Therefore, the designer of Java decided to define and implement the native `clone` method in the `Object` class.

Third, why doesn't the `Object` class implement the `Cloneable` interface? The answer is the same as in the first question.

Fourth, what would happen if the `House` class did not implement `Cloneable` in line 1 of Listing 13.11? `house1.clone()` would return `null` because `super.clone()` in line 28 would throw a `CloneNotSupportedException`.

Fifth, you may implement the `clone` method in the `House` class without invoking the clone method in the `Object` class as follows:

```
public Object clone() {
  // Perform a shallow copy
  House houseClone = new House(id, area);

  // Deep copy on whenBuilt
  houseClone.whenBuilt = new Date();
  houseClone.getWhenBuilt().setTime(whenBuilt.getTime());

  return houseClone;
}
```

In this case, the `House` class does not need to implement the `Cloneable` interface, and you have to make sure all the data fields are copied correctly. Using the `clone()` method in the `Object` class relieves you from manually copying the data fields. The `clone` method in the `Object` class automatically performs a shallow copy of all the data fields.

Check Point

13.7.1 Can a class invoke `super.clone()` when implementing the `clone()` method if the class does not implement `java.lang.Cloneable`? Does the `Date` class implement `Cloneable`?

13.7.2 What would happen if the `House` class (defined in Listing 13.11) did not override the `clone()` method or if `House` did not implement `java.lang.Cloneable`?

13.7.3 Show the output of the following code:

```
java.util.Date date = new java.util.Date();
java.util.Date date1 = date;
java.util.Date date2 = (java.util.Date)(date.clone());
System.out.println(date == date1);
System.out.println(date == date2);
System.out.println(date.equals(date2));
```

13.7.4 Show the output of the following code:

```
ArrayList<String> list = new ArrayList<>();
list.add("New York");
ArrayList<String> list1 = list;
ArrayList<String> list2 = (ArrayList<String>)(list.clone());
list.add("Atlanta");
System.out.println(list == list1);
System.out.println(list == list2);
System.out.println("list is " + list);
System.out.println("list1 is " + list1);
System.out.println("list2.get(0) is " + list2.get(0));
System.out.println("list2.size() is " + list2.size());
```

13.7.5 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    GeometricObject x = new Circle(3);
    GeometricObject y = x.clone();
    System.out.println(x == y);
  }
}
```

13.7.6 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    House house1 = new House(1, 1750, 50);
    House house2 = (House)house1.clone();
    System.out.println(house1.equals(house2);
  }
}
```

13.8 Interfaces vs. Abstract Classes

A class can implement multiple interfaces, but it can only extend one superclass.

Key Point

An interface can be used more or less the same way as an abstract class, but defining an interface is different from defining an abstract class. Table 13.2 summarizes the differences.

TABLE 13.2 Interfaces vs. Abstract Classes

	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	May contain public abstract instance methods, public default, and public static methods.

Java allows only *single inheritance* for class extension, but allows *multiple extensions* for interfaces. For example,

single inheritance
multiple inheritance

```
public class NewClass extends BaseClass
    implements Interface1, ... , InterfaceN {
  ...
}
```

An interface can inherit other interfaces using the `extends` keyword. Such an interface is called a *subinterface*. For example, `NewInterface` in the following code is a subinterface of `Interface1`, . . . , and `InterfaceN`.

subinterface

```
public interface NewInterface extends Interface1, ... , InterfaceN {
  // constants and abstract methods
}
```

A class implementing `NewInterface` must implement the abstract methods defined in `NewInterface`, `Interface1`, . . . , and `InterfaceN`. An interface can extend other interfaces, but not classes. A class can extend its superclass and implement multiple interfaces.

All classes share a single root, the `Object` class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class implements an interface, the interface is like a superclass for the class. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa. For example, suppose `c` is an instance of `Class2` in Figure 13.7. `c` is also an instance of `Object`, `Class1`, `Interface1`, `Interface1_1`, `Interface1_2`, `Interface2_1`, and `Interface2_2`.

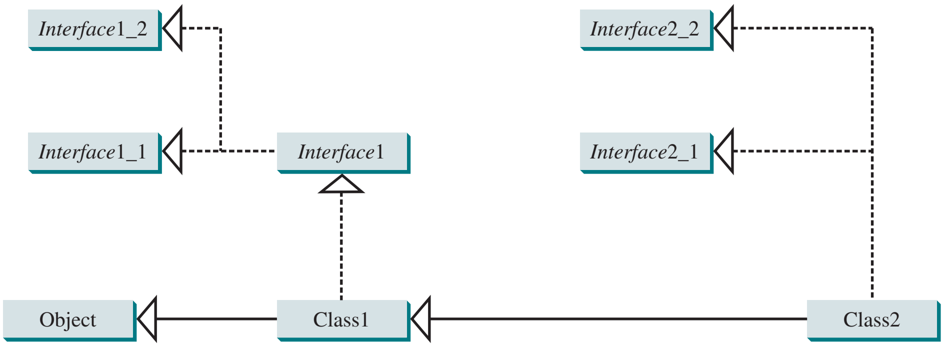


Figure 13.7 `Class1` implements `Interface1`; `Interface1` extends `Interface1_1` and `Interface1_2`. `Class2` extends `Class1` and implements `Interface2_1` and `Interface2_2`.

naming convention

Note

Class names are nouns. Interface names may be adjectives or nouns.

Design Guide

is-a relationship
is-kind-of relationship

Abstract classes and interfaces can both be used to specify common behavior of objects. How do you decide whether to use an interface or a class? In general, a *strong is-a relationship* that clearly describes a parent–child relationship should be modeled using classes. For example, Gregorian calendar is a calendar, so the relationship between the class `java.util.GregorianCalendar` and `java.util.Calendar` is modeled using class inheritance. A *weak is-a relationship*, also known as an *is-kind-of relationship*, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the `String` class implements the `Comparable` interface.

interface preferred

In general, interfaces are preferred over abstract classes because an interface can define a common supertype for unrelated classes. Interfaces are more flexible than classes. Consider the `Animal` class. Suppose the `howToEat` method is defined in the `Animal` class as follows:

`Animal` class

```
abstract class Animal {
  public abstract String howToEat();
}
```

Two subclasses of `Animal` are defined as follows:

`Chicken` class

```
class Chicken extends Animal {
  @Override
  public String howToEat() {
    return "Fry it";
  }
}
```

`Duck` class

```
class Duck extends Animal {
  @Override
  public String howToEat() {
    return "Roast it";
  }
}
```

Given this inheritance hierarchy, polymorphism enables you to hold a reference to a `Chicken` object or a `Duck` object in a variable of type `Animal`, as in the following code:

```
public static void main(String[] args) {
  Animal animal = new Chicken();
  eat(animal);

  animal = new Duck();
  eat(animal);
}

public static void eat(Animal animal) {
  System.out.println(animal.howToEat());
}
```

The JVM dynamically decides which `howToEat` method to invoke based on the actual object that invokes the method.

You can define a subclass of `Animal`. However, there is a restriction: the subclass must be for another animal (e.g., `Turkey`). Another issue arises: if an animal (e.g., `Tiger`) is not edible, it will not be appropriate to extend the `Animal` class.

Interfaces don't have these problems. Interfaces give you more flexibility than classes because you don't have to make everything fit into one type of class. You may define the `howToEat()` method in an interface, and let it serve as a common supertype for other classes. For example, see the following code:

```
public class DesignDemo {
  public static void main(String[] args) {
    Edible stuff = new Chicken();
    eat(stuff);

    stuff = new Duck();
    eat(stuff);

    stuff = new Broccoli();
    eat(stuff);
  }

  public static void eat(Edible stuff) {
    System.out.println(stuff.howToEat()):
  }
}
```

`Edible` interface

```
interface Edible {
  public String howToEat();
}
```

`Chicken` class

```
class Chicken implements Edible {
  @Override
  public String howToEat() {
    return "Fry it";
  }
}
```

`Duck` class

```
class Duck implements Edible {
  @Override
  public String howToEat() {
    return "Roast it";
  }
}
```

`Broccoli` class

```
class Broccoli implements Edible {
  @Override
  public String howToEat() {
    return "Stir-fry it";
  }
}
```

To define a class that represents edible objects, simply let the class implement the `Edible` interface. The class is now a subtype of the `Edible` type, and any `Edible` object can be passed to invoke the `howToEat` method.

Check Point

13.8.1 Give an example to show why interfaces are preferred over abstract classes.

13.8.2 Define the terms abstract classes and interfaces. What are the similarities and differences between abstract classes and interfaces?

13.8.3 True or false?

a. An interface is compiled into a separate bytecode file.
b. An interface can have static methods.
c. An interface can extend one or more interfaces.
d. An interface can extend an abstract class.
e. An interface can have default methods.

13.9 Case Study: The `Rational` Class

Key Point

This section shows how to design the `Rational` *class for representing and processing rational numbers.*

A rational number has a numerator and a denominator in the form `a/b`, where `a` is the numerator and `b` the denominator. For example, `1/3`, `3/4`, and `10/4` are rational numbers.

A rational number cannot have a denominator of `0`, but a numerator of `0` is fine. Every integer `i` is equivalent to a rational number `i/1`. Rational numbers are used in exact computations involving fractions—for example, `1/3 = 0.33333`. . . . This number cannot be precisely represented in floating-point format using either the data type `double` or `float`. To obtain the exact result, we must use rational numbers.

Java provides data types for integers and floating-point numbers, but not for rational numbers. This section shows how to design a class to represent rational numbers.

Since rational numbers share many common features with integers and floating-point numbers, and `Number` is the root class for numeric wrapper classes, it is appropriate to define `Rational` as a subclass of `Number`. Since rational numbers are comparable, the `Rational` class should also implement the `Comparable` interface. Figure 13.8 illustrates the `Rational` class and its relationship to the `Number` class and the `Comparable` interface.

A rational number consists of a numerator and a denominator. There are many equivalent rational numbers—for example, `1/3 = 2/6 = 3/9 = 4/12`. The numerator and the denominator of `1/3` have no common divisor except `1`, so `1/3` is said to be in *lowest terms*.

To reduce a rational number to its lowest terms, you need to find the greatest common divisor (GCD) of the absolute values of its numerator and denominator, then divide both the numerator and denominator by this value. You can use the method for computing the GCD of

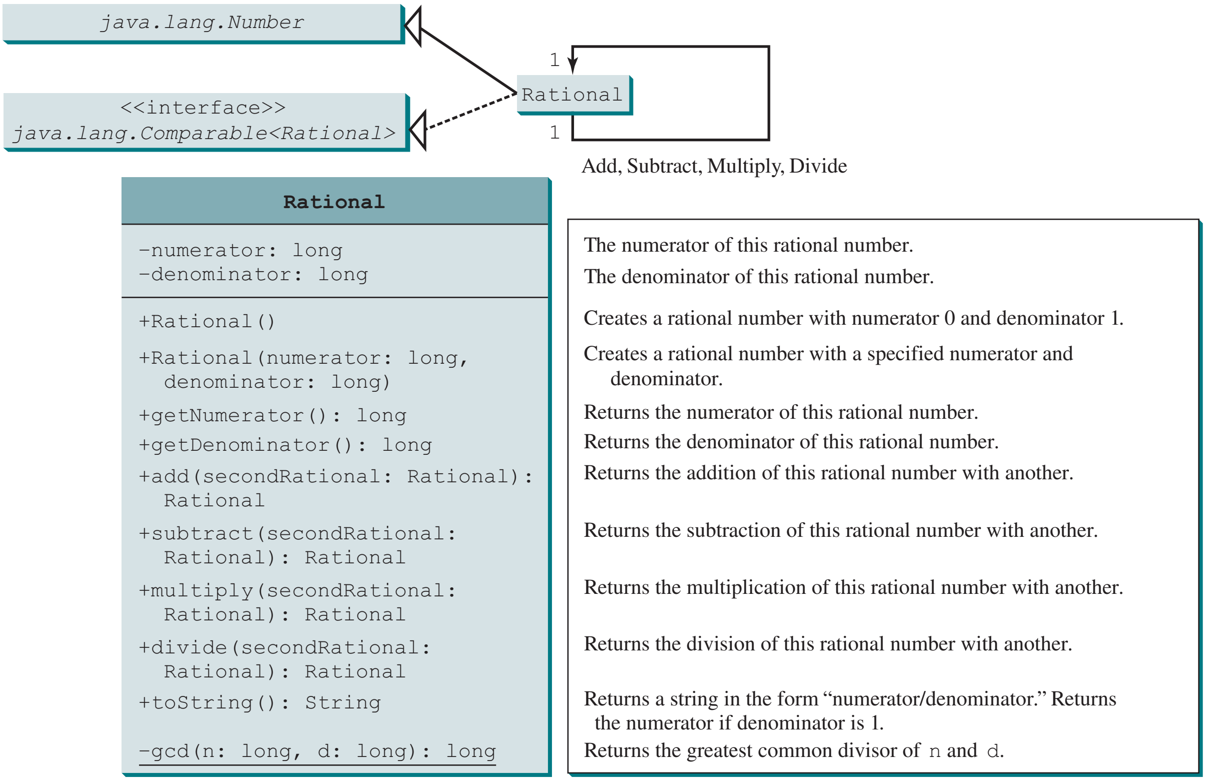

java.lang.Number
<<interface>>
java.lang.Comparable<Rational>
1
Rational
1
Add, Subtract, Multiply, Divide
Rational
-numerator: long
-denominator: long
+Rational()
+Rational(numerator: long, denominator: long)
+getNumerator(): long
+getDenominator(): long
+add(secondRational: Rational): Rational
+subtract(secondRational: Rational): Rational
+multiply(secondRational: Rational): Rational
+divide(secondRational: Rational): Rational
+toString(): String
-gcd(n: long, d: long): long
The numerator of this rational number.
The denominator of this rational number.
Creates a rational number with numerator 0 and denominator 1.
Creates a rational number with a specified numerator and denominator.
Returns the numerator of this rational number.
Returns the denominator of this rational number.
Returns the addition of this rational number with another.
Returns the subtraction of this rational number with another.
Returns the multiplication of this rational number with another.
Returns the division of this rational number with another.
Returns a string in the form "numerator/denominator." Returns the numerator if denominator is 1.
Returns the greatest common divisor of n and d.

FIGURE 13.8 The properties, constructors, and methods of the **Rational** class are illustrated in UML.

two integers **n** and **d**, as suggested in Listing 5.9, GreatestCommonDivisor.java. The numerator and denominator in a **Rational** object are reduced to their lowest terms.

As usual, let us first write a test program to create two **Rational** objects and test its methods. Listing 13.12 is a test program.

LISTING 13.12 TestRationalClass.java

```java
public class TestRationalClass {
  /** Main method */
  public static void main(String[] args) {
    // Create and initialize two rational numbers r1 and r2
    Rational r1 = new Rational(4, 2);
    Rational r2 = new Rational(2, 3);

    // Display results
    System.out.println(r1 + " + " + r2 + " = " + r1.add(r2));
    System.out.println(r1 + " - " + r2 + " = " + r1.subtract(r2));
    System.out.println(r1 + " * " + r2 + " = " + r1.multiply(r2));
    System.out.println(r1 + " / " + r2 + " = " + r1.divide(r2));
    System.out.println(r2 + " is " + r2.doubleValue());
  }
}
```

create a Rational

create a Rational

add

```
2 + 2/3 = 8/3
2 - 2/3 = 4/3
2 * 2/3 = 4/3
2 / 2/3 = 3
2/3 is 0.6666666666666666
```

The `main` method creates two rational numbers, `r1` and `r2` (lines 5 and 6), and displays the results of `r1 + r2`, `r1 - r2`, `r1 x r2`, and `r1 / r2` (lines 9–12). To perform `r1 + r2`, invoke `r1.add(r2)` to return a new `Rational` object. Similarly, invoke `r1.subtract(r2)` for `r1 - r2`, `r1.multiply(r2)` for `r1 x r2`, and `r1.divide(r2)` for `r1 / r2`.

The `doubleValue()` method displays the double value of `r2` (line 13). The `doubleValue()` method is defined in `java.lang.Number` and overridden in `Rational`.

Note when a string is concatenated with an object using the plus sign (`+`), the object's string representation from the `toString()` method is used to concatenate with the string. Thus, `r1 + " + " + r2 + " = " + r1.add(r2)` is equivalent to `r1.toString() + " + " + r2.toString() + " = " + r1.add(r2).toString()`.

The `Rational` class is implemented in Listing 13.13.

LISTING 13.13 Rational.java

```java
public class Rational extends Number implements Comparable<Rational> {
  // Data fields for numerator and denominator
  private long numerator = 0;
  private long denominator = 1;

  /** Construct a rational with default properties */
  public Rational() {
    this(0, 1);
  }

  /** Construct a rational with specified numerator and denominator */
  public Rational(long numerator, long denominator) {
    long gcd = gcd(numerator, denominator);
    this.numerator = (denominator > 0 ? 1 : -1) * numerator / gcd;
    this.denominator = Math.abs(denominator) / gcd;
  }

  /** Find GCD of two numbers */
  private static long gcd(long n, long d) {
    long n1 = Math.abs(n);
    long n2 = Math.abs(d);
    int gcd = 1;

    for (int k = 1; k <= n1 && k <= n2; k++) {
      if (n1 % k == 0 && n2 % k == 0)
        gcd = k;
    }

    return gcd;
  }

  /** Return numerator */
  public long getNumerator() {
    return numerator;
  }

  /** Return denominator */
  public long getDenominator() {
    return denominator;
  }

  /** Add a rational number to this rational */
  public Rational add(Rational secondRational) {
    long n = numerator * secondRational.getDenominator() +
      denominator * secondRational.getNumerator();
```

$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$

```
    long d = denominator * secondRational.getDenominator();
    return new Rational(n, d);
  }

```

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

```
  /** Subtract a rational number from this rational */
  public Rational subtract(Rational secondRational) {
    long n = numerator * secondRational.getDenominator()
      - denominator * secondRational.getNumerator();
    long d = denominator * secondRational.getDenominator();
    return new Rational(n, d);
  }

```

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

```
  /** Multiply a rational number by this rational */
  public Rational multiply(Rational secondRational) {
    long n = numerator * secondRational.getNumerator();
    long d = denominator * secondRational.getDenominator();
    return new Rational(n, d);
  }

```

$$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$$

```
  /** Divide a rational number by this rational */
  public Rational divide(Rational secondRational) {
    long n = numerator * secondRational.getDenominator();
    long d = denominator * secondRational.numerator;
    return new Rational(n, d);
  }

  @Override
  public String toString() {
    if (denominator == 1)
      return numerator + "";
    else
      return numerator + "/" + denominator;
  }

  @Override // Override the equals method in the Object class
  public boolean equals(Object other) {
    if ((this.subtract((Rational)(other))).getNumerator() == 0)
      return true;
    else
      return false;
  }

  @Override // Implement the abstract intValue method in Number
  public int intValue() {
    return (int)doubleValue();
  }

  @Override // Implement the abstract floatValue method in Number
  public float floatValue() {
    return (float)doubleValue();
  }

  @Override // Implement the doubleValue method in Number
  public double doubleValue() {
    return numerator *  1.0 / denominator;
  }

  @Override // Implement the abstract longValue method in Number
  public long longValue() {
    return (long)doubleValue();
```

```
106    }
107
108    @Override // Implement the compareTo method in Comparable
109    public int compareTo(Rational o) {
110      if (this.subtract(o).getNumerator() > 0)
111        return 1;
112      else if (this.subtract(o).getNumerator() < 0)
113        return -1;
114      else
115        return 0;
116    }
117  }
```

The rational number is encapsulated in a `Rational` object. Internally, a rational number is represented in its lowest terms (line 13) and the numerator determines its sign (line 14). The denominator is always positive (line 15).

The `gcd` method (lines 19–30 in the `Rational` class) is private; it is not intended for use by clients. The `gcd` method is only for internal use by the `Rational` class. The `gcd` method is also static, since it is not dependent on any particular `Rational` object.

The `abs(x)` method (lines 20 and 21 in the `Rational` class) is defined in the `Math` class and returns the absolute value of `x`.

Two `Rational` objects can interact with each other to perform add, subtract, multiply, and divide operations. These methods return a new `Rational` object (lines 43–70). The math formula for performing these operations are as follows:

$$a/b + c/d = (ad + bc)/(bd) \text{ (e.g., } 2/3 + 3/4 = (2*4 + 3*3)/(3*4) = 17/12)$$

$$a/b - c/d = (ad - bc)/(bd) \text{ (e.g., } 2/3 - 3/4 = (2*4 - 3*3)/(3*4) = -1/12)$$

$$a/b * c/d = (ac)/(bd) \text{ (e.g., } 2/3*3/4 = (2*3)/(3*4) = 6/12 = 1/2)$$

$$(a/b) / (c/d) = (ad)/(bc) \text{ (e.g., } (2/3) / (3/4) = (2*4)/(3*3) = 8/9)$$

The methods `toString` and `equals` in the `Object` class are overridden in the `Rational` class (lines 72–86). The `toString()` method returns a string representation of a `Rational` object in the form `numerator/denominator`, or simply `numerator` if `denominator` is `1`. The `equals(Object other)` method returns true if this rational number is equal to the other rational number.

The abstract methods `intValue`, `longValue`, `floatValue`, and `doubleValue` in the `Number` class are implemented in the `Rational` class (lines 88–106). These methods return the `int`, `long`, `float`, and `double` value for this rational number.

The `compareTo(Rational other)` method in the `Comparable` interface is implemented in the `Rational` class (lines 108–116) to compare this rational number to the other rational number.

immutable

Note

The getter methods for the properties `numerator` and `denominator` are provided in the `Rational` class, but the setter methods are not provided, so, once a `Rational` object is created, its contents cannot be changed. The `Rational` class is immutable. The `String` class and the wrapper classes for primitive-type values are also immutable.

encapsulation

Note

The numerator and denominator are represented using two variables. It is possible to use an array of two integers to represent the numerator and denominator (see Programming Exercise 13.14). The signatures of the public methods in the `Rational` class are not changed, although the internal representation of a rational number is changed. This is a good example to illustrate the idea that the data fields of a class should be kept private so as to encapsulate the implementation of the class from the use of the class.

overflow

The `Rational` class has serious limitations and can easily overflow. For example, the following code will display an incorrect result, because the denominator is too large:

```
public class Test {
  public static void main(String[] args) {
    Rational r1 = new Rational(1, 123456789);
    Rational r2 = new Rational(1, 123456789);
    Rational r3 = new Rational(1, 123456789);
    System.out.println("r1 * r2 * r3 is " +
      r1.multiply(r2.multiply(r3)));
  }
}
```

```
r1 * r2 * r3 is -1/2204193661661244627
```

To fix it, you can implement the `Rational` class using the `BigInteger` for numerator and denominator (see Programming Exercise 13.15).

Check Point

13.9.1 Show the output of the following code:

```
Rational r1 = new Rational(-2, 6);
System.out.println(r1.getNumerator());
System.out.println(r1.getDenominator());
System.out.println(r1.intValue());
System.out.println(r1.doubleValue());
```

13.9.2 Why is the following code wrong?

```
Rational r1 = new Rational(-2, 6);
Object r2 = new Rational(1, 45);
System.out.println(r2.compareTo(r1));
```

13.9.3 Why is the following code wrong?

```
Object r1 = new Rational(-2, 6);
Rational r2 = new Rational(1, 45);
System.out.println(r2.compareTo(r1));
```

13.9.4 Simplify the code in lines 82–85 in Listing 13.13, Rational.java, using one line of code without using the if statement. Simplify the code in lines 110–115 using a conditional operator.

13.9.5 Trace the program carefully and show the output of the following code:

```
Rational r1 = new Rational(1, 2);
Rational r2 = new Rational(1, -2);
System.out.println(r1.add(r2));
```

13.9.6 The preceding question shows a bug in the `toString` method. Revise the `toString()` method to fix the error.

13.10 Class-Design Guidelines

Class-design guidelines are helpful for designing sound classes.

Key Point

You have learned how to design classes from the preceding example and from many other examples in the previous chapters. This section summarizes some of the guidelines.

13.10.1 Cohesion

A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff are different entities.

coherent purpose

separate responsibilities

A single entity with many responsibilities can be broken into several classes to separate the responsibilities. The classes `String`, `StringBuilder`, and `StringBuffer` all deal with strings, for example, but have different responsibilities. The `String` class deals with immutable strings, the `StringBuilder` class is for creating mutable strings, and the `StringBuffer` class is similar to `StringBuilder`, except that `StringBuffer` contains synchronized methods for updating strings.

13.10.2 Consistency

naming conventions

Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. A popular style is to place the data declaration before the constructor, and place constructors before methods.

naming consistency

Make the names consistent. It is not a good practice to choose different names for similar operations. For example, the `length()` method returns the size of a `String`, a `StringBuilder`, and a `StringBuffer`. It would be inconsistent if different names were used for this method in these classes.

no-arg constructor

In general, you should consistently provide a public no-arg constructor for constructing a default instance. If a class does not support a no-arg constructor, document the reason. If no constructors are defined explicitly, a public default no-arg constructor with an empty body is assumed.

If you want to prevent users from creating an object for a class, you can declare a private constructor in the class, as is the case for the `Math` class and the `GuessDate` class.

13.10.3 Encapsulation

encapsulate data fields

A class should use the `private` modifier to hide its data from direct access by clients. This makes the class easy to maintain.

Provide a getter method only if you want the data field to be readable and provide a setter method only if you want the data field to be updateable. For example, the `Rational` class provides a getter method for `numerator` and `denominator`, but no setter method, because a `Rational` object is immutable.

13.10.4 Clarity

Cohesion, consistency, and encapsulation are good guidelines for achieving design clarity. In addition, a class should have a clear contract that is easy to explain and easy to understand.

easy to explain

Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on how or when the user can use it, design the properties in a way that lets the user set them in any order and with any combination of values, and design methods that function independently of their order of occurrence. For example, the `Loan` class contains the properties `loanAmount`, `numberOfYears`, and `annualInterestRate`. The values of these properties can be set in any order.

independent methods

intuitive meaning

Methods should be defined intuitively without causing confusion. For example, the `substring(int beginIndex, int endIndex)` method in the `String` class is somewhat confusing. The method returns a substring from `beginIndex` to `endIndex – 1`, rather than to `endIndex`. It would be more intuitive to return a substring from `beginIndex` to `endIndex`.

independent properties

You should not declare a data field that can be derived from other data fields. For example, the following `Person` class has two data fields: `birthDate` and `age`. Since `age` can be derived from `birthDate`, `age` should not be declared as a data field.

```
public class Person {
  private java.util.Date birthDate;
  private int age;

  ...
}
```

13.10.5 Completeness

Classes are designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through properties and methods. For example, the `String` class contains more than 40 methods that are useful for a variety of applications.

13.10.6 Instance vs. Static

A variable or method that is dependent on a specific instance of the class must be an instance variable or method. A variable that is shared by all the instances of a class should be declared static. For example, the variable `numberOfObjects` in `Circle` in Listing 9.8 is shared by all the objects of the `Circle` class, and therefore is declared static. A method that is not dependent on a specific instance should be defined as a static method. For instance, the `getNumberOfObjects()` method in `Circle` is not tied to any specific instance and therefore is defined as a static method.

Always reference static variables and methods from a class name (rather than a reference variable) to improve readability and avoid errors.

Do not pass a parameter from a constructor to initialize a static data field. It is better to use a setter method to change the static data field. Thus, the following class in (a) is better replaced by (b):

```java
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1, int t2) {
    ...
  }
}
```

(a)

```java
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1) {
    ...
  }

  public static void setT2(int t2) {
    SomeThing.t2 = t2;
  }
}
```

(b)

Instance and static are integral parts of object-oriented programming. A data field or method is either instance or static. Do not mistakenly overlook static data fields or methods. It is a common design error to define an instance method that should have been static. For example, the `factorial(int n)` method for computing the factorial of `n` should be defined static because it is independent of any specific instance.

common design error

A constructor is always instance because it is used to create a specific instance. A static variable or method can be invoked from an instance method, but an instance variable or method cannot be invoked from a static method.

13.10.7 Inheritance vs. Aggregation

The difference between inheritance and aggregation is the difference between an is-a and a has-a relationship. For example, an apple is a fruit; thus, you would use inheritance to model the relationship between the classes `Apple` and `Fruit`. A person has a name; thus, you would use aggregation to model the relationship between the classes `Person` and `Name`.

13.10.8 Interfaces vs. Abstract Classes

Both interfaces and abstract classes can be used to specify common behavior for objects. How do you decide whether to use an interface or a class? In general, a strong is-a relationship that clearly describes a parent–child relationship should be modeled using classes. For example, since an orange is a fruit, their relationship should be modeled using class inheritance. A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the `String` class implements the `Comparable` interface. A circle or a rectangle is a geometric object, so `Circle` can be designed as a subclass of `GeometricObject`. Circles are different and comparable based on their radii, so `Circle` can implement the `Comparable` interface.

Interfaces are more flexible than abstract classes because a subclass can extend only one superclass, but can implement any number of interfaces. However, interfaces cannot contain data fields. In Java 8, interfaces can contain default methods and static methods, which are very useful to simplify class design. We will give examples of this type of design in Chapter 20, Lists, Stacks, Queues, and Priority Queues.

13.10.1 Describe class-design guidelines.

Check Point

KEY TERMS

abstract class	500
abstract method	500
deep copy	520
interface	523
marker interface	518
shallow copy	520
subinterface	523

CHAPTER SUMMARY

1. *Abstract classes* are like regular classes with data and methods, but you cannot create instances of abstract classes using the `new` operator.

2. An *abstract method* cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the inherited abstract methods of the superclass, the subclass must be defined as abstract.

3. A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn't contain any abstract methods.

4. A subclass can be abstract even if its superclass is concrete.

5. An *interface* is a class-like construct that contains only constants, abstract methods, default methods, and static methods. In many ways, an interface is similar to an abstract class, but an abstract class can contain data fields.

6. An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class.

7. The `java.lang.Comparable` interface defines the `compareTo` method. Many classes in the Java library implement `Comparable`.

8. The `java.lang.Cloneable` interface is a *marker interface*. An object of the class that implements the `Cloneable` interface is cloneable.

9. A class can extend only one superclass but can implement one or more interfaces.

10. An interface can extend one or more interfaces.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

Sections 13.2 and 13.3

****13.1** (*Triangle class*) Design a new `Triangle` class that extends the abstract `GeometricObject` class. Draw the UML diagram for the classes `Triangle` and `GeometricObject` then implement the `Triangle` class. Write a test program that prompts the user to enter three sides of the triangle, a color, and a Boolean value to indicate whether the triangle is filled. The program should create a `Triangle` object with these sides, and set the `color` and `filled` properties using the input. The program should display the area, perimeter, color, and true or false to indicate whether it is filled or not.

***13.2** (*Shuffle ArrayList*) Write the following method that shuffles an `ArrayList` of numbers:

```
public static void shuffle(ArrayList<Number> list)
```

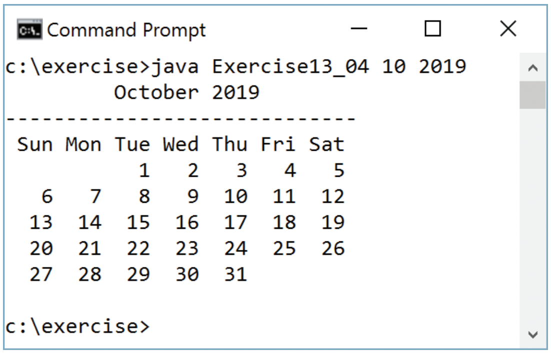
***13.3** (*Sort ArrayList*) Write the following method that sorts an `ArrayList` of numbers:

```
public static void sort(ArrayList<Number> list)
```

****13.4** (*Display calendars*) Rewrite the `PrintCalendar` class in Listing 6.12 to display a calendar for a specified month using the `Calendar` and `GregorianCalendar` classes. Your program receives the month and year from the command line. For example:

```
java Exercise13_04 5 2016
```

This displays the calendar shown in Figure 13.9.



```
c:\exercise>java Exercise13_04 10 2019
         October 2019
-----------------------------
 Sun Mon Tue Wed Thu Fri Sat
           1   2   3   4   5
   6   7   8   9  10  11  12
  13  14  15  16  17  18  19
  20  21  22  23  24  25  26
  27  28  29  30  31

c:\exercise>
```

FIGURE 13.9 The program displays a calendar for May 2016.

You can also run the program without the year. In this case, the year is the current year. If you run the program without specifying a month and a year, the month is the current month.

Sections 13.4–13.8v

***13.5** (*Enable `GeometricObject` comparable*) Modify the `GeometricObject` class to implement the `Comparable` interface and define a static `max` method in the `GeometricObject` class for finding the larger of two `GeometricObject` objects. Draw the UML diagram and implement the new `GeometricObject` class. Write a test program that uses the `max` method to find the larger of two circles, the larger of two rectangles.

***13.6** (*The `ComparableCircle` class*) Define a class named `ComparableCircle` that extends `Circle` and implements `Comparable`. Draw the UML diagram and implement the `compareTo` method to compare the circles on the basis of area. Write a test class to find the larger of two instances of `ComparableCircle` objects, and the larger between a circle and a rectangle.

***13.7** (*The `Colorable` interface*) Design an interface named `Colorable` with a `void` method named `howToColor()`. Every class of a colorable object must implement the `Colorable` interface. Design a class named `Square` that extends `GeometricObject` and implements `Colorable`. Implement `howToColor` to display the message `Color all four sides`. The `Square` class contains a data field `side` with getter and setter methods, and a constructor for constructing a `Square` with a specified side. The `Square` class has a private double data field named `side` with its getter and setter methods. It has a no-arg constructor to create a `Square` with side 0, and another constructor that creates a `Square` with the specified side.

Draw a UML diagram that involves `Colorable`, `Square`, and `GeometricObject`. Write a test program that creates an array of five `GeometricObject`s. For each object in the array, display its area and invoke its `howToColor` method if it is colorable.

***13.8** (*Revise the `MyStack` class*) Rewrite the `MyStack` class in Listing 11.10 to perform a deep copy of the `list` field.

***13.9** (*Enable `Circle` comparable*) Rewrite the `Circle` class in Listing 13.2 to extend `GeometricObject` and implement the `Comparable` interface. Override the `equals` method in the `Object` class. Two `Circle` objects are equal if their radii are the same. Draw the UML diagram that involves `Circle`, `GeometricObject`, and `Comparable`.

***13.10** (*Enable `Rectangle` comparable*) Rewrite the `Rectangle` class in Listing 13.3 to extend `GeometricObject` and implement the `Comparable` interface. Override the `equals` method in the `Object` class. Two `Rectangle` objects are equal if their areas are the same. Draw the UML diagram that involves `Rectangle`, `GeometricObject`, and `Comparable`.

***13.11** (*The `Octagon` class*) Write a class named `Octagon` that extends `GeometricObject` and implements the `Comparable` and `Cloneable` interfaces. Assume all eight sides of the octagon are of equal length. The area can be computed using the following formula:

$$area = \left(2 + 4/\sqrt{2}\right) * side * side$$

The `Octagon` class has a private double data field named side with its getter and setter methods. The class has a no-arg constructor that creates an `Octagon` with side 0, and a constructor to create an `Octagon` with a specified side.

Draw the UML diagram that involves `Octagon`, `GeometricObject`, `Comparable`, and `Cloneable`. Write a test program that creates an `Octagon` object with side value `5` and displays its area and perimeter. Create a new object using the `clone` method, and compare the two objects using the `compareTo` method.

***13.12** (*Sum the areas of geometric objects*) Write a method that sums the areas of all the geometric objects in an array. The method signature is

```
public static double sumArea(GeometricObject[] a)
```

Write a test program that creates an array of four objects (two circles and two rectangles) and computes their total area using the `sumArea` method.

***13.13** (*Enable the `Course` class cloneable*) Rewrite the `Course` class in Listing 10.6 to add a `clone` method to perform a deep copy on the `students` field.

Section 13.9

***13.14** (*Demonstrate the benefits of encapsulation*) Rewrite the `Rational` class in Listing 13.13 using a new internal representation for the numerator and denominator. Create an array of two integers as follows:

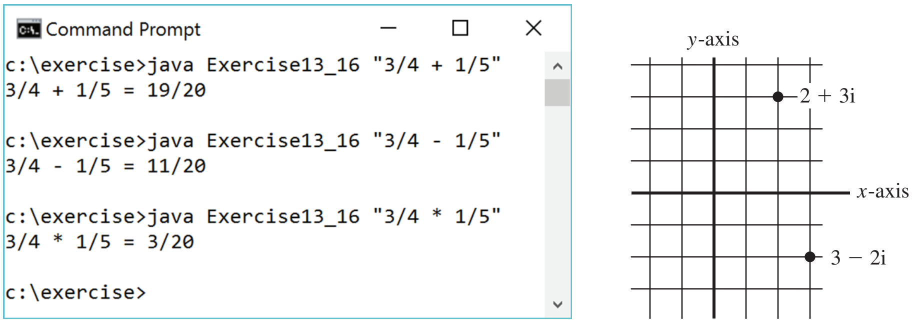
```
private long[] r = new long[2];
```

Use `r[0]` to represent the numerator and `r[1]` to represent the denominator. The signatures of the methods in the `Rational` class are not changed, so a client application that uses the previous `Rational` class can continue to use this new `Rational` class without being recompiled.

***13.15** (*Use `BigInteger` for the `Rational` class*) Redesign and implement the `Rational` class in Listing 13.13 using `BigInteger` for the numerator and denominator. Write a test program that prompts the user to enter two rational numbers and display the results as shown in the following sample run:

```
Enter the first rational number: 3 454 ↵Enter
Enter the second rational number: 7 2389 ↵Enter
3/454 + 7/2389 = 10345/1084606
3/454 – 7/2389 = 3989/1084606
3/454 * 7/2389 = 21/1084606
3/454 / 7/2389 = 7167/3178
7/2389 is 0.002930096274591879З
```

***13.16** (*Create a rational-number calculator*) Write a program similar to Listing 7.9, Calculator.java. Instead of using integers, use rationals, as shown in Figure 13.10. You will need to use the `split` method in the `String` class, introduced in Section 10.10.3, Replacing and Splitting Strings, to retrieve the numerator string and denominator string, and convert strings into integers using the `Integer.parseInt` method.



```
c:\exercise>java Exercise13_16 "3/4 + 1/5"
3/4 + 1/5 = 19/20

c:\exercise>java Exercise13_16 "3/4 - 1/5"
3/4 - 1/5 = 11/20

c:\exercise>java Exercise13_16 "3/4 * 1/5"
3/4 * 1/5 = 3/20

c:\exercise>
```

(a) (b)

FIGURE 13.10 (a) The program takes a string argument that consists of operand1, operator, and operand2 from the command line and displays the expression and the result of the arithmetic operation. (b) A complex number can be interpreted as a point in a plane.

***13.17** (_Math: The_ `Complex` _class_) A complex number is a number in the form $a + bi$, where a and b are real numbers and i is $\sqrt{-1}$. The numbers `a` and `b` are known as the real part and imaginary part of the complex number, respectively. You can perform addition, subtraction, multiplication, and division for complex numbers using the following formulas:

$$a + bi + c + di = (a + c) + (b + d)i$$

$$a + bi - (c + di) = (a - c) + (b - d)i$$

$$(a + bi) * (c + di) = (ac - bd) + (bc + ad)i$$

$$(a + bi)/(c + di) = (ac + bd)/(c^2 + d^2) + (bc - ad)i/(c^2 + d^2)$$

You can also obtain the absolute value for a complex number using the following formula:

$$|a + bi| = \sqrt{a^2 + b^2}$$

(A complex number can be interpreted as a point on a plane by identifying the (a,b) values as the coordinates of the point. The absolute value of the complex number corresponds to the distance of the point to the origin, as shown in Figure 13.10.)

Design a class named `Complex` for representing complex numbers and the methods `add`, `subtract`, `multiply`, `divide`, and `abs` for performing complex-number operations, and override `toString` method for returning a string representation for a complex number. The `toString` method returns `(a + bi)` as a string. If `b` is `0`, it simply returns `a`. Your `Complex` class should also implement `Cloneable` and `Comparable`. Compare two complex numbers using their absolute values.

Provide three constructors `Complex(a, b)`, `Complex(a)`, and `Complex()`. `Complex()` creates a `Complex` object for number `0`, and `Complex(a)` creates a `Complex` object with `0` for `b`. Also provide the `getRealPart()` and `getImaginaryPart()` methods for returning the real part and the imaginary part of the complex number, respectively.

Draw the UML class diagram and implement the class. Use the code at https://liveexample.pearsoncmg.com/test/Exercise13_17.txt to test your implementation. Here is a sample run:

```
Enter the first complex number: 3.5 5.5 Enter
Enter the second complex number: -3.5 1 Enter
(3.5 + 5.5i) + (-3.5 + 1.0i) = 0.0 + 6.5i
(3.5 + 5.5i) - (-3.5 + 1.0i) = 7.0 + 4.5i
(3.5 + 5.5i) * (-3.5 + 1.0i) = -17.75 + -15.75i
(3.5 + 5.5i) / (-3.5 + 1.0i) = -0.5094 + -1.7i
|(3.5 + 5.5i)| = 6.519202405202649
false
3.5
5.5
[-3.5 + 1.0i, 4.0 + -0.5i, 3.5 + 5.5i, 3.5 + 5.5i]
```

13.18 (*Use the* **`Rational`** *class*) Write a program that computes the following summation series using the **`Rational`** class:

$$\frac{1}{2} + \frac{2}{3} + \frac{3}{4} + \cdots + \frac{98}{99} + \frac{99}{100}$$

You will discover that the output is incorrect because of integer overflow (too large). To fix this problem, see Programming Exercise 13.15.

13.19 (*Convert decimals to fractions*) Write a program that prompts the user to enter a decimal number and displays the number in a fraction. (*Hint*: read the decimal number as a string, extract the integer part and fractional part from the string, and use the **`Rational`** class in Listing 13.13 to obtain a rational number for the decimal number.) Here are some sample runs:

```
Enter a decimal number: 3.25 Enter
The fraction number is 13/4
```

```
Enter a decimal number: -0.45452 Enter
The fraction number is -11363/25000
```

13.20 (*Algebra: solve quadratic equations*) Rewrite Programming Exercise 3.1 to obtain imaginary roots if the determinant is less than 0 using the **`Complex`** class in Programming Exercise 13.17. Here are some sample runs:

```
Enter a, b, c: 1 3 1 Enter
The roots are -0.381966 and -2.61803
```

```
Enter a, b, c: 1 2 1 Enter
The root is -1
```

```
Enter a, b, c: 1 2 3 ↵Enter
The roots are -1.0 + 1.4142i and -1.0 + -1.4142i
```

13.21 (*Algebra: vertex form equations*) The equation of a parabola can be expressed in either standard form ($y = ax^2 + bx + c$) or vertex form ($y = a(x - h)^2 + k$). Write a program that prompts the user to enter a, b, and c as integers in standard form and displays $h\left(= \frac{-b}{2a}\right)$ and $k\left(= \frac{4ac - b^2}{4a}\right)$ in the vertex form. Display h and k as rational numbers. Here are some sample runs:

```
Enter a, b, c: 1 3 1 ↵Enter
h is -3/2 k is -5/4
```

```
Enter a, b, c: 2 3 4 ↵Enter
h is -3/4 k is 23/8
```

CHAPTER

14

JAVAFX BASICS

Objectives

- To distinguish between JavaFX, Swing, and AWT (§14.2).
- To write a simple JavaFX program and understand the relationship among stages, scenes, and nodes (§14.3).
- To create user interfaces using panes, groups, UI controls, and shapes (§14.4).
- To update property values automatically through property binding (§14.5).
- To use the common properties **`style`** and **`rotate`** for nodes (§14.6).
- To create colors using the **`Color`** class (§14.7).
- To create fonts using the **`Font`** class (§14.8).
- To create images using the **`Image`** class, and to create image views using the **`ImageView`** class (§14.9).
- To layout nodes using **`Pane`**, **`StackPane`**, **`FlowPane`**, **`GridPane`**, **`BorderPane`**, **`HBox`**, and **`VBox`** (§14.10).
- To display text using the **`Text`** class, and create shapes using the **`Line`**, **`Circle`**, **`Rectangle`**, **`Ellipse`**, **`Arc`**, **`Polygon`**, and **`Polyline`** classes (§14.11).
- To develop the reusable GUI component **`ClockPane`** for displaying an analog clock (§14.12).

14.1 Introduction

Key Point

JavaFX is an excellent pedagogical tool for learning object-oriented programming.

JavaFX is a new framework for developing Java GUI programs. The JavaFX API is an excellent example of how the object-oriented principles are applied. This chapter serves two purposes. First, it presents the basics of JavaFX programming. Second, it uses JavaFX to demonstrate object-oriented design and programming. Specifically, this chapter introduces the framework of JavaFX and discusses JavaFX GUI components and their relationships. You will learn how to develop simple GUI programs using layout panes, groups, buttons, labels, text fields, colors, fonts, images, image views, and shapes.

14.2 JavaFX vs. Swing and AWT

Key Point

Swing and AWT are replaced by the JavaFX platform for developing rich GUI applications.

AWT

Swing

JavaFX

When Java was introduced, the GUI classes were bundled in a library known as the *Abstract Windows Toolkit* (*AWT*). AWT is fine for developing simple graphical user interfaces, but not for developing comprehensive GUI projects. In addition, AWT is prone to platform-specific bugs. The AWT user-interface components were replaced by a more robust, versatile, and flexible library known as *Swing*. Swing components are painted directly on canvases using Java code. Swing components depend less on the target platform, and use less of the native GUI resources. Swing is designed for developing desktop GUI applications. It is now replaced by a completely new GUI platform known as *JavaFX*. JavaFX incorporates modern GUI technologies to enable you to develop rich GUI applications. In addition, JavaFX provides a multitouch support for touch-enabled devices such as tablets and smart phones. JavaFX has a built-in 2D, 3D, animation support, and video and audio playback. Using third-party software, you can develop JavaFX programs to be deployed on devices running iOS or Android.

why teaching JavaFX

This book teaches Java GUI programming using JavaFX for three reasons. First, JavaFX is much simpler to learn and use for new Java programmers. Second, JavaFX is a better pedagogical tool for demonstrating object-oriented programming than Swing. Third, Swing is essentially dead because it will not receive any further enhancement. JavaFX is the new GUI tool for developing cross-platform rich GUI applications on desktop computers and on handheld devices.

Check Point

14.2.1 Explain the evolution of Java GUI technologies.

14.2.2 Explain why this book teaches Java GUI using JavaFX.

14.3 The Basic Structure of a JavaFX Program

Key Point

The `javafx.application.Application` class defines the essential framework for writing JavaFX programs.

VideoNote

Getting started with JavaFX

We begin by writing a simple JavaFX program that illustrates the basic structure of a JavaFX program. Every JavaFX program is defined in a class that extends `javafx.application.Application`, as shown in Listing 14.1.

LISTING 14.1 `MyJavaFX.java`

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;
```

```
 5
 6  public class MyJavaFX extends Application {
 7    @Override // Override the start method in the Application class
 8    public void start(Stage primaryStage) {
 9      // Create a scene and place a button in the scene
10      Button btOK = new Button("OK");
11      Scene scene = new Scene(btOK, 200, 250);
12      primaryStage.setTitle("MyJavaFX"); // Set the stage title
13      primaryStage.setScene(scene); // Place the scene in the stage
14      primaryStage.show(); // Display the stage
15    }
16
17    /**
18     * The main method is only needed for the IDE with limited
19     * JavaFX support. Not needed for running from the command line.
20     */
21    public static void main(String[] args) {
22      Application.launch(args);
23    }
24  }
```

extend Application
override start
create a button
create a scene
set stage title
set a scene
display stage
main method
launch application

JavaFX on NetBeans and Eclipse

You can test and run your program from a command window or from an IDE such as NetBeans or Eclipse. A sample run of the program is shown in Figure 14.1. Since JDK 11, JavaFX has become a separate module. Supplements II.F–H give the tips for running JavaFX programs from a command window, NetBeans, and Eclipse using JDK 11.

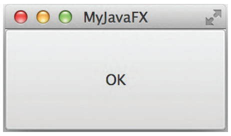


Figure 14.1 A simple JavaFX displays a button in the window.

`launch`

The `launch` method (line 22) is a static method defined in the `Application` class for launching a stand-alone JavaFX application. The `main` method (lines 21–23) is not needed if you run the program from the command line. It may be needed to launch a JavaFX program from an IDE with a limited JavaFX support. When you run a JavaFX application without a main method, JVM automatically invokes the `launch` method to run the application.

construct application

start application

The main class overrides the `start` method defined in `javafx.application.Application` (line 8). After a JavaFX application is launched, the JVM constructs an instance of the class using its `no-arg` constructor and invokes its `start` method. The `start` method normally places UI controls in a scene and displays the scene in a stage, as shown in Figure 14.2a.

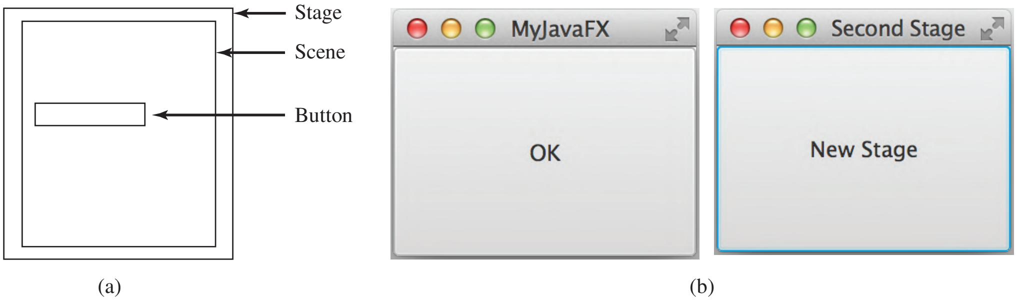
`scene`

Line 10 creates a `Button` object and places it in a `Scene` object (line 11). A `Scene` object can be created using the constructor `Scene(node, width, height)`. This constructor specifies the width and height of the scene and places the node in the scene.

primary stage

A `Stage` object is a window. A `Stage` object called *primary stage* is automatically created by the JVM when the application is launched. Line 13 sets the scene to the primary stage and line 14 displays the primary stage. JavaFX names the `Stage` and `Scene` classes using the analogy from the theater. You may think of stage as the platform to support scenes, and nodes as actors to perform in the scenes.

You can create additional stages if needed. The JavaFX program in Listing 14.2 displays two stages, as shown in Figure 14.2b.



(a) (b)

FIGURE 14.2 (a) Stage is a window for displaying a scene that contains nodes. (b) Multiple stages can be displayed in a JavaFX program.

LISTING 14.2 MultipleStageDemo.java

primary stage in start

display primary stage

create second stage

display second stage

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class MultipleStageDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place a button in the scene
    Scene scene = new Scene(new Button("OK"), 200, 250);
    primaryStage.setTitle("MyJavaFX"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    Stage stage = new Stage(); // Create a new stage
    stage.setTitle("Second Stage"); // Set the stage title
    // Set a scene with a button in the stage
    stage.setScene(new Scene(new Button("New Stage"), 200, 250));
    stage.show(); // Display the stage
  }
}
```

main method omitted

Note the main method is omitted in the listing since it is identical for every JavaFX application. From now on, we will not list the `main` method in our JavaFX source code for brevity.

prevent stage resizing

By default, the user can resize the stage. To prevent the user from resizing the stage, invoke `stage.setResizable(false)`.

Check Point

14.3.1 How do you define a JavaFX main class? What is the signature of the `start` method? What is a stage? What is a primary stage? Is a primary stage automatically created? How do you display a stage? Can you prevent the user from resizing the stage? Can you replace `Application.launch(args)` by `launch(args)` in line 22 in Listing 14.1?

14.3.2 Show the output of the following JavaFX program:

```java
import javafx.application.Application;
import javafx.stage.Stage;

public class Test extends Application {
  public Test() {
    System.out.println("Test constructor is invoked");
  }
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    System.out.println("start method is invoked");
  }

  public static void main(String[] args) {
    System.out.println("launch application");
    Application.launch(args);
  }
}
```

14.4 Panes, Groups, UI Controls, and Shapes

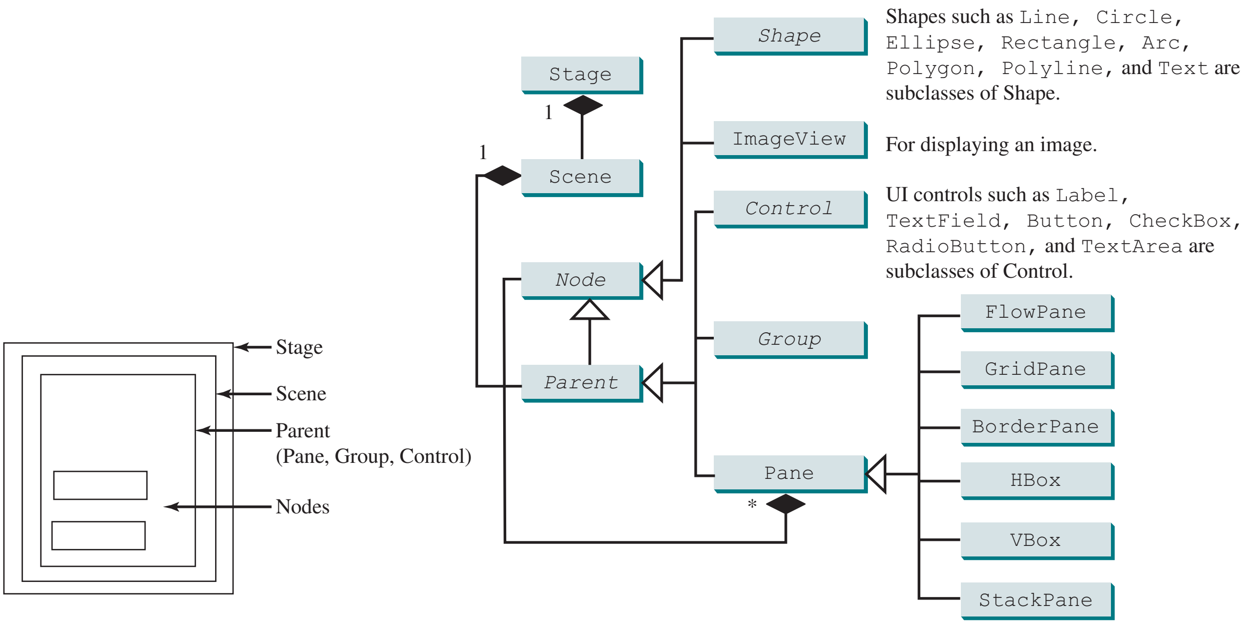
Panes, Groups, UI controls, and shapes are subtypes of **`Node`**.

Key Point

When you run MyJavaFX in Listing 14.1, the window is displayed as shown in Figure 14.1. The button is always centered in the scene and occupies the entire window no matter how you resize it. You can fix the problem by setting the position and size properties of a button. However, a better approach is to use container classes, called *panes*, for automatically laying out the nodes in a desired location and size. You place nodes inside a pane then place the pane into a scene. A *node* is a visual component such as a shape, an image view, a UI control, a group, or a pane. A *shape* refers to a text, line, circle, ellipse, rectangle, arc, polygon, polyline, and so on. A *UI control* refers to a label, button, check box, radio button, text field, text area, and so on. A group is a container that groups a collection of nodes. You can apply transformations or effects to a group, which automatically apply to all the children in the group. A scene can be displayed in a stage, as shown in Figure 14.3a. The relationship among **`Stage`**, **`Scene`**, **`Node`**, **`Control`**, **`Group`**, and **`Pane`** is illustrated in the UML diagram, as shown in Figure 14.3b. Note a **`Scene`** can contain a **`Control`**, **`Group`**, or a **`Pane`**, but not a **`Shape`** or an **`ImageView`**. A **`Pane`** or a **`Group`** can contain any subtype of **`Node`**. You can create a **`Scene`** using the constructor **`Scene(Parent, width, height)`** or **`Scene(Parent)`**. The dimension of the scene is automatically decided in the latter constructor. Every subclass of **`Node`** has a no-arg constructor for creating a default node.

pane
node
shape
UI control
group

Listing 14.3 gives a program that places a button in a pane, as shown in Figure 14.4.



Shapes such as `Line`, `Circle`, `Ellipse`, `Rectangle`, `Arc`, `Polygon`, `Polyline`, and `Text` are subclasses of Shape.

For displaying an image.

UI controls such as `Label`, `TextField`, `Button`, `CheckBox`, `RadioButton`, and `TextArea` are subclasses of Control.

(a) (b)

FIGURE 14.3 (a) Panes and groups are used to hold nodes. (b) Nodes can be shapes, image views, UI controls, groups, and panes.

LISTING 14.3 ButtonInPane.java

create a pane
add a button
add pane to scene
display stage
main method omitted

```
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.control.Button;
 4 import javafx.stage.Stage;
 5 import javafx.scene.layout.StackPane;
 6
 7 public class ButtonInPane extends Application {
 8   @Override // Override the start method in the Application class
 9   public void start(Stage primaryStage) {
10     // Create a scene and place a button in the scene
11     StackPane pane = new StackPane();
12     pane.getChildren().add(new Button("OK"));
13     Scene scene = new Scene(pane, 200, 50);
14     primaryStage.setTitle("Button in a pane"); // Set the stage title
15     primaryStage.setScene(scene); // Place the scene in the stage
16     primaryStage.show(); // Display the stage
17   }
18 }
```

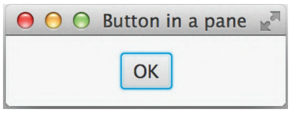


FIGURE 14.4 A button is placed in the center of the pane.

`ObservableList`

The program creates a **`StackPane`** (line 11) and adds a button as a child of the pane (line 12). The **`getChildren()`** method returns an instance of **`javafx.collections.ObservableList`**. **`ObservableList`** behaves very much like an **`ArrayList`** for storing a collection of elements. Invoking **`add(e)`** adds an element to the list. The **`StackPane`** places the nodes in the center of the pane on top of each other. Here, there is only one node in the pane. The **`StackPane`** respects a node's preferred size. Therefore, you see the button displayed in its preferred size.

Along with many other constructors, each pane and group have a no-arg constructor, and also a constructor that adds one or more children to the pane or group. Thus, the code in lines 11 and 12 can be replaced using one statement:

```
StackPane pane = new StackPane(new Button("OK"));
```

Listing 14.4 gives an example that displays a circle in the center of the pane, as shown in Figure 14.5a.

LISTING 14.4 ShowCircle.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ShowCircle extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
```

```
11         // Create a circle and set its properties
12         Circle circle = new Circle();
13         circle.setCenterX(100);
14         circle.setCenterY(100);
15         circle.setRadius(50);
16         circle.setStroke(Color.BLACK);
17         circle.setFill(Color.WHITE);
18
19         // Create a pane to hold the circle
20         Pane pane = new Pane();
21         pane.getChildren().add(circle);
22
23         // Create a scene and place it in the stage
24         Scene scene = new Scene(pane, 200, 200);
25         primaryStage.setTitle("ShowCircle"); // Set the stage title
26         primaryStage.setScene(scene); // Place the scene in the stage
27         primaryStage.show(); // Display the stage
28     }
29   }
```

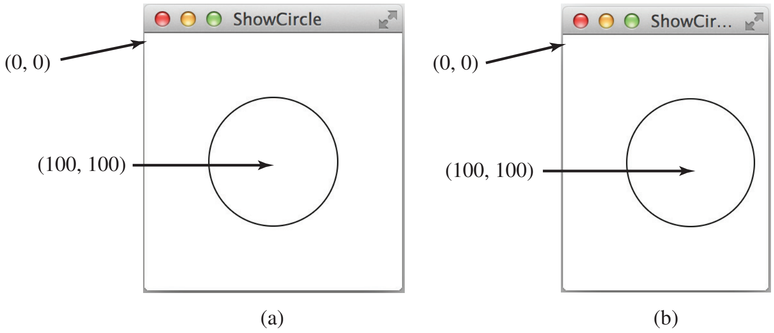
create a circle
set circle properties

create a pane
add circle to pane

add pane to scene

display stage

main method omitted



(a) (b)

Figure 14.5 (a) A circle is displayed in the center of the scene. (b) The circle is not centered after the window is resized.

The program creates a **`Circle`** (line 12) and sets its center at (100, 100) (lines 13 and 14), which is also the center for the scene, since the scene is created with the width and height of 200 (line 24). The radius of the circle is set to 50 (line 15). Note the measurement units for graphics in Java are all in *pixels*.

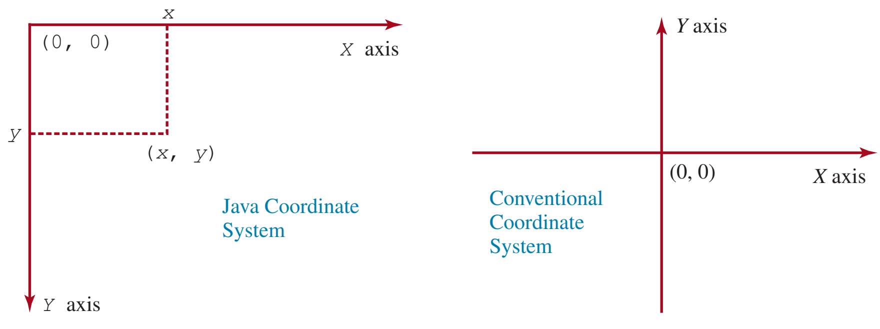
pixels

The stroke color (i.e., the color to draw the circle) is set to black (line 16). The fill color (i.e., the color to fill the circle) is set to white (line 17). You may set the color to **`null`** to specify that no color is set.

set color

The program creates a **`Pane`** (line 20) and places the circle in the pane (line 21). Note the coordinates of the upper-left corner of the pane is (**`0`**, **`0`**) in the Java coordinate system, as shown in Figure 14.6a, as opposed to the conventional coordinate system, where (**`0`**, **`0`**) is at the center of the window, as shown in Figure 14.6b. The *x*-coordinate increases from left to right, and the *y*-coordinate increases downward in the Java coordinate system.

The pane is placed in the scene (line 24) and the scene is set in the stage (line 26). The circle is displayed in the center of the stage, as shown in Figure 14.5a. However, if you resize the window, the circle is not centered, as shown in Figure 14.5b. In order to display the circle centered as the window resizes, the *x*- and *y*-coordinates of the circle center need to be reset to the center of the pane. This can be done by using property binding, introduced in the next section.



(a) (b)

FIGURE 14.6 The Java coordinate system is measured in pixels, with `(0, 0)` at its upper-left corner.

Check Point

14.4.1 How do you create a `Scene` object? How do you set a scene in a stage? How do you place a circle into a scene?

14.4.2 What is a pane? What is a node? How do you place a node in a pane? Can you directly place a `Shape` or an `ImageView` into a `Scene`? Can you directly place a `Control` or a `Pane` into a `Scene`?

14.4.3 How do you create a `Circle`? How do you set its center location and radius? How do you set its stroke color and fill color?

14.4.4 How do you replace the code in lines 20 and 21 in Listing 14.4 using one statement?

14.5 Property Binding

Key Point

You can bind a target object to a source object. A change in the source object will be automatically reflected in the target object.

target object
source object
binding object
binding property
bindable object
observable object

JavaFX introduces a new concept called *property binding* that enables a *target object* to be bound to a *source object*. If the value in the source object changes, the target object is also automatically changed. The target object is called a *binding object* or a *binding property*, and the source object is called a *bindable object* or *observable object*. As discussed in Listing 14.4, ShowCircle.java, the circle is not centered after the window is resized. In order to display the circle centered as the window resizes, the *x*- and *y*-coordinates of the circle center need to be reset to the center of the pane. This can be done by binding the `centerX` with pane's `width/2` and `centerY` with pane's `height/2`, as given in lines 16–17 Listing 14.5.

VideoNote
Understand property binding

LISTING 14.5 ShowCircleCentered.java

```
 1  import javafx.application.Application;
 2  import javafx.scene.Scene;
 3  import javafx.scene.layout.Pane;
 4  import javafx.scene.paint.Color;
 5  import javafx.scene.shape.Circle;
 6  import javafx.stage.Stage;
 7
 8  public class ShowCircleCentered extends Application {
 9    @Override // Override the start method in the Application class
10    public void start(Stage primaryStage) {
```

```
11     // Create a pane to hold the circle
12     Pane pane = new Pane();
13 
14     // Create a circle and set its properties
15     Circle circle = new Circle();
16     circle.centerXProperty().bind(pane.widthProperty().divide(2));
17     circle.centerYProperty().bind(pane.heightProperty().divide(2));
18     circle.setRadius(50);
19     circle.setStroke(Color.BLACK);
20     circle.setFill(Color.WHITE);
21     pane.getChildren().add(circle); // Add circle to the pane
22 
23     // Create a scene and place it in the stage
24     Scene scene = new Scene(pane, 200, 200);
25     primaryStage.setTitle("ShowCircleCentered"); // Set the stage title
26     primaryStage.setScene(scene); // Place the scene in the stage
27     primaryStage.show(); // Display the stage
28   }
29 }
```

create a pane

create a circle
bind properties

add circle to pane

add pane to scene

display stage

main method omitted

The **`Circle`** class has the **`centerX`** property for representing the *x*-coordinate of the circle center. This property like many properties in JavaFX classes can be used both as target and source in a property binding. A binding property is an object that can be bound to a source object. A target listens to the changes in the source and automatically updates itself once a change is made in the source. A target binds with a source using the **`bind`** method as follows:

```
target.bind(source);
```

the `Property` interface

the `ObservableValue` interface

The **`bind`** method is defined in the **`javafx.beans.property.Property`** interface. A binding property is an instance of **`javafx.beans.property.Property`**. An observable source object is an instance of the **`javafx.beans.value.ObservableValue`** interface. An **`ObservableValue`** is an entity that wraps a value and allows to observe the value for changes.

common binding properties

common `ObservableValue` objects

A binding property is an object. JavaFX defines binding properties for primitive types and strings. For a **`double`**/**`float`**/**`long`**/**`int`**/**`boolean`** value, its binding property type is **`DoubleProperty`**/**`FloatProperty`**/**`LongProperty`**/**`IntegerProperty`**/**`BooleanProperty`**, respectively. For a string, its binding property type is **`StringProperty`**. These properties are also subtypes of **`ObservableValue`**. Therefore, they can be used as both source and target in a binding.

value getter method
value setter method
property getter method

By convention, each binding property (e.g., **`centerX`**) in a JavaFX class (e.g., **`Circle`**) has a getter (e.g., **`getCenterX()`**) and setter (e.g., **`setCenterX(double)`**) method for returning and setting the property's value. It also has a getter method for returning the property itself. The naming convention for this method is the property name followed by the word **`Property`**. For example, the property getter method for **`centerX`** is **`centerXProperty()`**. We call the **`getCenterX()`** method as the *value getter method*, the **`setCenterX(double)`** method as the *value setter method*, and **`centerXProperty()`** as the *property getter method*. Note **`getCenterX()`** returns a **`double`** value, and **`centerXProperty()`** returns an object of the **`DoubleProperty`** type. Figure 14.7a shows the convention for defining a binding property in a class, and Figure 14.7b shows a concrete example in which **`centerX`** is a binding property of the type **`DoubleProperty`**.

The program in Listing 14.5 is the same as in Listing 14.4 except that it binds **`circle`**'s **`centerX`** and **`centerY`** properties to half of **`pane`**'s width and height (lines 16 and 17). Note **`circle.centerXProperty()`** returns **`centerX`** and **`pane.widthProperty()`** returns

```
public class SomeClassName {
  private PropertyType x;

  /** Value getter method */
  public propertyValueType getX() { ... }

  /** Value setter method */
  public void setX(propertyValueType value) { ... }

  /** Property getter method */
  public PropertyType xProperty() { ... }

}
```

(a) `x` is a binding property

```
public class Circle {
  private DoubleProperty centerX;

  /** Value getter method */
  public double getCenterX() { ... }

  /** Value setter method */
  public void setCenterX(double value) { ... }

  /** Property getter method */
  public DoubleProperty centerXProperty() { ... }
}
```

(b) `centerX` is binding property in the `Circle` class

FIGURE 14.7 A binding property has a value getter method, setter method, and property getter method.

`width`. Both `centerX` and `width` are binding properties of the `DoubleProperty` type. The numeric binding property classes such as `DoubleProperty` and `IntegerProperty` contain the `add`, `subtract`, `multiply`, and `divide` methods for adding, subtracting, multiplying, and dividing a value in a binding property and returning a new observable property. Therefore, `pane.widthProperty().divide(2)` returns a new observable property that represents half of the `pane`'s width. The statement

```
circle.centerXProperty().bind(pane.widthProperty().divide(2));
```

is the same as

```
DoubleProperty centerX = circle.centerXProperty();
DoubleProperty width = pane.widthProperty();
centerX.bind(width.divide(2));
```

Since `centerX` is bound to `width.divide(2)`, when `pane`'s width is changed, `centerX` automatically updates itself to match `pane`'s width / 2.

Listing 14.6 gives another example that demonstrates bindings.

LISTING 14.6 `BindingDemo.java`

```
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

public class BindingDemo {
  public static void main(String[] args) {
    DoubleProperty d1 = new SimpleDoubleProperty(1);
    DoubleProperty d2 = new SimpleDoubleProperty(2);
    d1.bind(d2);
    System.out.println("d1 is " + d1.getValue()
      + " and d2 is " + d2.getValue());
    d2.setValue(70.2);
    System.out.println("d1 is " + d1.getValue()
      + " and d2 is " + d2.getValue());
  }
}
```

create a `DoubleProperty`
create a `DoubleProperty`
bind property
set a new source value

```
d1 is 2.0 and d2 is 2.0
d1 is 70.2 and d2 is 70.2
```

The program creates an instance of `DoubleProperty` using `SimpleDoubleProperty(1)` (line 6). Note that `DoubleProperty`, `FloatProperty`, `LongProperty`, `IntegerProperty`, and `BooleanProperty` are abstract classes. Their concrete subclasses `SimpleDoubleProperty`, `SimpleFloatProperty`, `SimpleLongProperty`, `SimpleIntegerProperty`, and `SimpleBooleanProperty` are used to create instances of these properties. These classes are very much like wrapper classes `Double`, `Float`, `Long`, `Integer`, and `Boolean` with additional features for property binding.

The program binds `d1` with `d2` (line 8). Now the values in `d1` and `d2` are the same. After setting `d2` to `70.2` (line 11), `d1` also becomes `70.2` (line 13).

unidirectional binding

bidirectional binding

The binding demonstrated in this example is known as *unidirectional binding*. Occasionally, it is useful to synchronize two properties so a change in one property is reflected in another object, and vice versa. This is called a *bidirectional binding*. If the target and source are both binding properties and observable properties, they can be bound bidirectionally using the `bindBidirectional` method.

Check Point

14.5.1 What is a binding property? What interface defines a binding property? What interface defines a source object? What are the binding object types for `int`, `long`, `float`, `double`, and `boolean`? Are `Integer` and `Double` binding properties? Can `Integer` and `Double` be used as source objects in a binding?

14.5.2 Following the JavaFX binding property naming convention, for a binding property named `age` of the `IntegerProperty` type, what is its value getter method, value setter method, and property getter method?

14.5.3 Can you create an object of `IntegerProperty` using `new IntegerProperty(3)`? If not, what is the correct way to create it? What will be the output if line 8 is replaced by `d1.bind(d2.multiply(2))` in Listing 14.6? What will be the output if line 8 is replaced by `d1.bind(d2.add(2))` in Listing 14.6?

14.5.4 What is unidirectional binding and what is bidirectional binding? Are all binding properties capable of bidirectional binding? Write a statement to bind property `d1` with property `d2` bidirectionally.

14.6 Common Properties and Methods for Nodes

The **`Node`** *class defines many properties and methods that are common to all nodes.*

Key Point

Nodes share many common properties. This section introduces two such properties: `style` and `rotate`.

JavaFX CSS

JavaFX style properties are similar to cascading style sheets (CSS) used to specify the styles for HTML elements in a Web page. Therefore, the style properties in JavaFX are called *JavaFX CSS*. In JavaFX, a style property is defined with a prefix `–fx–`. Each node has its own style properties. You can find these properties at docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html. For information on HTML and CSS, see Supplements V.A and V.B. If you are not familiar with HTML and CSS, you can still use JavaFX CSS.

The syntax for setting a style is `styleName:value`. Multiple style properties for a node can be set together separated by semicolon (`;`). For example, the following statement:

setStyle

```
circle.setStyle("-fx-stroke: black; -fx-fill: red;");
```

sets two JavaFX CSS properties for a circle. This statement is equivalent to the following two statements:

```
circle.setStroke(Color.BLACK);
circle.setFill(Color.RED);
```

If an incorrect JavaFX CSS is used, your program will still compile and run, but the style will be ignored.

The **rotate** property enables you to specify an angle in degrees for rotating a node from its center. If the degree is positive, the rotation is performed clockwise; otherwise, it is performed counterclockwise. For example, the following code rotates a button 80 degrees:

```
button.setRotate(80);
```

Listing 14.7 gives an example that creates a button, sets its style, and adds it to a pane. It then rotates the pane 45 degrees and sets its style with border color red and background color light gray, as shown in Figure 14.8.

LISTING 14.7 NodeStyleRotateDemo.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;
import javafx.scene.layout.StackPane;

public class NodeStyleRotateDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place a button in the scene
    StackPane pane = new StackPane();
    Button btOK = new Button("OK");
    btOK.setStyle("-fx-border-color: blue;");
    pane.getChildren().add(btOK);

    pane.setRotate(45);
    pane.setStyle(
      "-fx-border-color: red; -fx-background-color: lightgray;");

    Scene scene = new Scene(pane, 200, 250);
    primaryStage.setTitle("NodeStyleRotateDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

rotate the pane
set style for pane

main method omitted

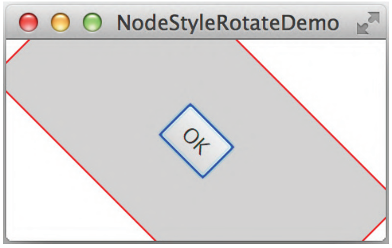


FIGURE 14.8 A pane's style is set and the pane is rotated 45 degrees.

As seen in Figure 14.8, rotating a pane causes all its containing nodes rotated as well.

contains method
setScaleX method
setScaleY method

The **Node** class contains many useful methods that can be applied to all nodes. For example, you can use the **contains(double x, double y)** method to test whether a point (*x*, *y*) is inside the boundary of a node and use the **setScaleX(double scale)** and **setScaleY(double scale)** methods to scale a node.

Check Point

14.6.1 How do you set a style of a node with border color red? Modify the code to set the text color for the button to red.

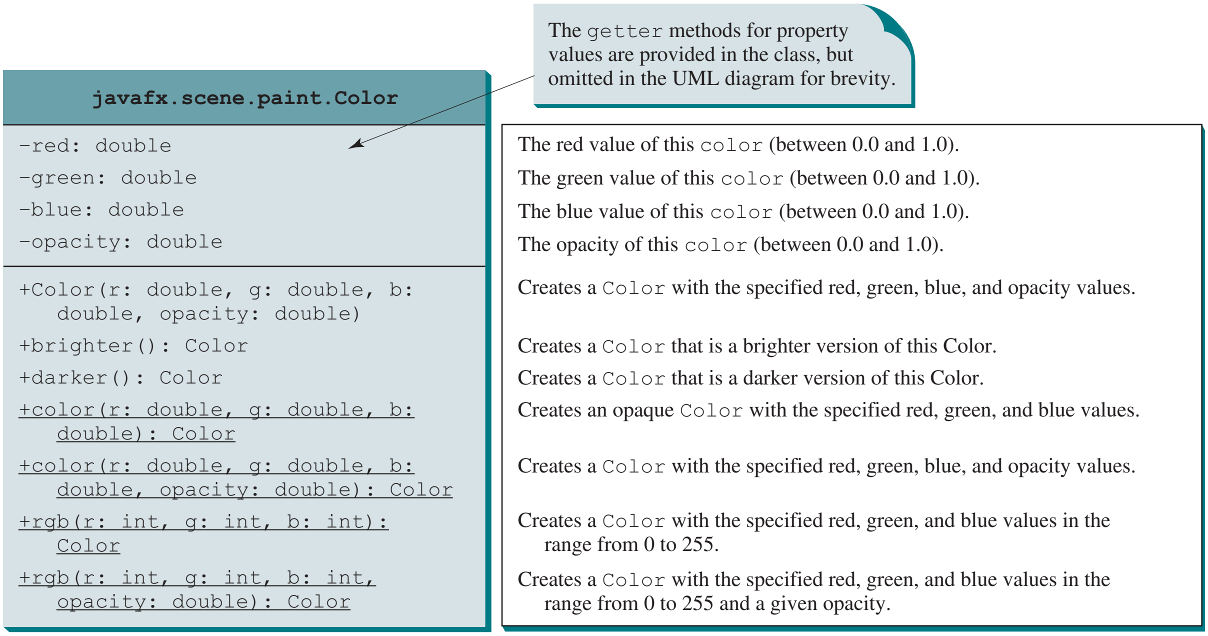
14.6.2 Can you rotate a pane, a text, or a button? Modify the code to rotate the button 15 degrees counterclockwise? How do you test if a point is inside a node? How do you scale up or down a node?

14.7 The `Color` Class

The **`Color`** *class can be used to create colors.*

Key Point

JavaFX defines the abstract `Paint` class for painting a node. The `javafx.scene.paint.Color` is a concrete subclass of `Paint`, which is used to encapsulate colors, as shown in Figure 14.9.



The `getter` methods for property values are provided in the class, but omitted in the UML diagram for brevity.

`javafx.scene.paint.Color`	
`-red: double`	The red value of this `color` (between 0.0 and 1.0).
`-green: double`	The green value of this `color` (between 0.0 and 1.0).
`-blue: double`	The blue value of this `color` (between 0.0 and 1.0).
`-opacity: double`	The opacity of this `color` (between 0.0 and 1.0).
`+Color(r: double, g: double, b: double, opacity: double)`	Creates a `Color` with the specified red, green, blue, and opacity values.
`+brighter(): Color`	Creates a `Color` that is a brighter version of this Color.
`+darker(): Color`	Creates a `Color` that is a darker version of this Color.
`+color(r: double, g: double, b: double): Color`	Creates an opaque `Color` with the specified red, green, and blue values.
`+color(r: double, g: double, b: double, opacity: double): Color`	Creates a `Color` with the specified red, green, blue, and opacity values.
`+rgb(r: int, g: int, b: int): Color`	Creates a `Color` with the specified red, green, and blue values in the range from 0 to 255.
`+rgb(r: int, g: int, b: int, opacity: double): Color`	Creates a `Color` with the specified red, green, and blue values in the range from 0 to 255 and a given opacity.

FIGURE 14.9 `Color` encapsulates information about colors.

A `color` instance can be constructed using the following constructor:

```
public Color(double r, double g, double b, double opacity);
```

in which `r`, `g`, and `b` specify a color by its red, green, and blue components with values in the range from `0.0` (darkest shade) to `1.0` (lightest shade). The `opacity` value defines the transparency of a color within the range from `0.0` (completely transparent) to `1.0` (completely opaque). This is known as the RGBA model, where RGBA stands for red, green, blue, and alpha. The alpha value indicates the opacity. For example,

RBGA model

```
Color color = new Color(0.25, 0.14, 0.333, 0.51);
```

For an interactive demo, see liveexample.pearsoncmg.com/dsanimation/FigureSection14_7.html.

The `Color` class is immutable. Once a `Color` object is created, its properties cannot be changed. The `brighter()` method returns a new `Color` with a larger red, green, and blue values, and the `darker()` method returns a new `Color` with a smaller red, green, and blue values. The `opacity` value is the same as in the original `Color` object.

You can also create a `Color` object using the static methods `color(r, g, b)`, `color(r, g, b, opacity)`, `rgb(r, g, b)`, and `rgb(r, g, b, opacity)`.

Alternatively, you can use one of the many standard colors such as `BEIGE`, `BLACK`, `BLUE`, `BROWN`, `CYAN`, `DARKGRAY`, `GOLD`, `GRAY`, `GREEN`, `LIGHTGRAY`, `MAGENTA`, `NAVY`, `ORANGE`, `PINK`, `RED`, `SILVER`, `WHITE`, and `YELLOW` defined as constants in the `Color` class. The following code, for instance, sets the fill color of a circle to red:

```
circle.setFill(Color.RED);
```

14.7.1 How do you create a color? What is wrong about creating a `Color` using `new Color(1.2, 2.3, 3.5, 4)`? Which of two colors is darker, `new Color(0, 0, 0, 1)` or `new Color(1, 1, 1, 1)`? Does invoking `c.darker()` change the color value in `c`?

14.7.2 How do you create a `Color` object with a random color?

14.7.3 How do you set a circle object `c` with blue fill color using the `setFill` method and the `setStyle` method?

14.8 The Font Class

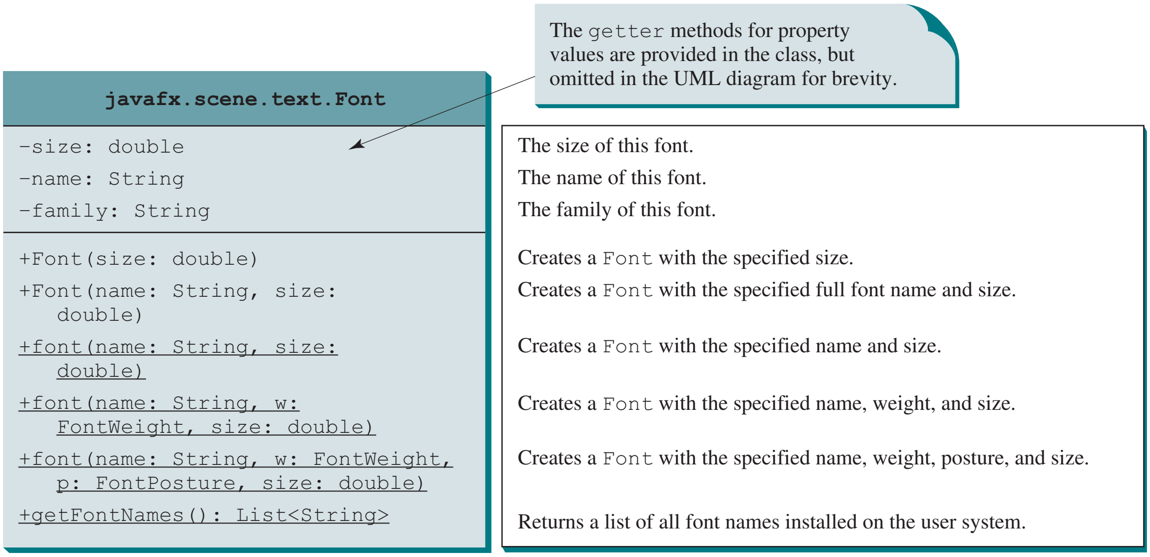
A `Font` *describes font name, weight, and size.*

You can set fonts for rendering the text. The `javafx.scene.text.Font` class is used to create fonts, as shown in Figure 14.10.

A `Font` instance can be constructed using its constructors or using its static methods. A `Font` is defined by its name, weight, posture, and size. Times New Roman, Courier, and Arial are examples of font names. You can obtain a list of available font family names by invoking the static `getFontNames()` method. This method returns `List<String>`. `List` is an interface that defines common methods for lists. `ArrayList`, introduced in Section 11.11, is a concrete class that implements `List`. There are two font postures defined as constants in the `FontPosture` class: `FontPosture.ITALIC` and `FontPosture.REGULAR`.

```
Font font1 = new Font("SansSerif", 16);
Font font2 = Font.font("Times New Roman", FontWeight.BOLD,
  FontPosture.ITALIC, 12);
```

Listing 14.8 gives a program that displays a label using the font (Times New Roman, bold, italic, and size 20), as shown in Figure 14.11.


javafx.scene.text.Font
-size: double
-name: String
-family: String
+Font(size: double)
+Font(name: String, size: double)
+font(name: String, size: double)
+font(name: String, w: FontWeight, size: double)
+font(name: String, w: FontWeight, p: FontPosture, size: double)
+getFontNames(): List<String>
The size of this font.
The name of this font.
The family of this font.
Creates a Font with the specified size.
Creates a Font with the specified full font name and size.
Creates a Font with the specified name and size.
Creates a Font with the specified name, weight, and size.
Creates a Font with the specified name, weight, posture, and size.
Returns a list of all font names installed on the user system.

The `getter` methods for property values are provided in the class, but omitted in the UML diagram for brevity.

FIGURE 14.10 `Font` encapsulates information about fonts.

LISTING 14.8 FontDemo.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
```

```
 5  import javafx.scene.shape.Circle;
 6  import javafx.scene.text.*;
 7  import javafx.scene.control.*;
 8  import javafx.stage.Stage;
 9
10  public class FontDemo extends Application {
11    @Override // Override the start method in the Application class
12    public void start(Stage primaryStage) {
13      // Create a pane to hold the circle
14      Pane pane = new StackPane();
15
16      // Create a circle and set its properties
17      Circle circle = new Circle();
18      circle.setRadius(50);
19      circle.setStroke(Color.BLACK);
20      circle.setFill(new Color(0.5, 0.5, 0.5, 0.1));
21      pane.getChildren().add(circle); // Add circle to the pane
22
23      // Create a label and set its properties
24      Label label = new Label("JavaFX");
25      label.setFont(Font.font("Times New Roman",
26        FontWeight.BOLD, FontPosture.ITALIC, 20));
27      pane.getChildren().add(label);
28
29      // Create a scene and place it in the stage
30      Scene scene = new Scene(pane);
31      primaryStage.setTitle("FontDemo"); // Set the stage title
32      primaryStage.setScene(scene); // Place the scene in the stage
33      primaryStage.show(); // Display the stage
34    }
35  }
```

create a `StackPane`

create a `Circle`

create a `Color`

add circle to the pane

create a `label`

create a `font`

add label to the pane

main method omitted

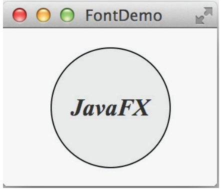


FIGURE 14.11 A label is on top of a circle displayed in the center of the scene.

The program creates a **StackPane** (line 14) and adds a circle and a label to it (lines 21 and 27). These two statements can be combined using the following one statement:

```
pane.getChildren().addAll(circle, label);
```

A **StackPane** places the nodes in the center and nodes are placed on top of each other. A custom color is created and set as a fill color for the circle (line 20). The program creates a label and sets a font (line 25) so that the text in the label is displayed in Times New Roman, bold, italic, and 20 pixels.

As you resize the window, the circle and label are displayed in the center of the window because the circle and label are placed in the stack pane. Stack pane automatically places nodes in the center of the pane.

A **Font** object is immutable. Once a **Font** object is created, its properties cannot be changed.

Check Point

14.8.1 How do you create a **Font** object with font name **Courier**, size **20**, and weight **bold**?

14.8.2 How do you find all available fonts on your system?

14.9 The `Image` and `ImageView` Classes

Key Point

The `Image` *class represents a graphical image, and the* `ImageView` *class can be used to display an image.*

VideoNote
Use `Image` and `ImageView`

The `javafx.scene.image.Image` class represents a graphical image and is used for loading an image from a specified filename or a URL. For example, `new Image("image/us.gif")` creates an `Image` object for the image file `us.gif` under the directory `image` in the Java class directory and `new Image("http://liveexample.pearsoncmg.com/book/image/us.gif")` creates an `Image` object for the image file in the URL on the Web.

The `javafx.scene.image.ImageView` is a node for displaying an image. An `ImageView` can be created from an `Image` object. For example, the following code creates an `ImageView` from an image file:

```
Image image = new Image("image/us.gif");
ImageView imageView = new ImageView(image);
```

Alternatively, you can create an `ImageView` directly from a file or a URL as follows:

```
ImageView imageView = new ImageView("image/us.gif");
```

The UML diagrams for the `Image` class and `ImageView` classes are illustrated in Figures 14.12 and 14.13.

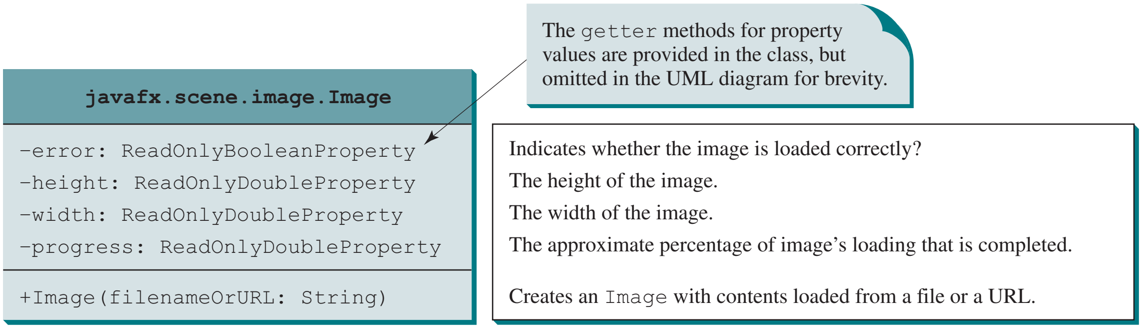

The getter methods for property values are provided in the class, but omitted in the UML diagram for brevity.
javafx.scene.image.Image
-error: ReadOnlyBooleanProperty
-height: ReadOnlyDoubleProperty
-width: ReadOnlyDoubleProperty
-progress: ReadOnlyDoubleProperty
+Image(filenameOrURL: String)
Indicates whether the image is loaded correctly?
The height of the image.
The width of the image.
The approximate percentage of image's loading that is completed.
Creates an Image with contents loaded from a file or a URL.

FIGURE 14.12 `Image` encapsulates information about images.

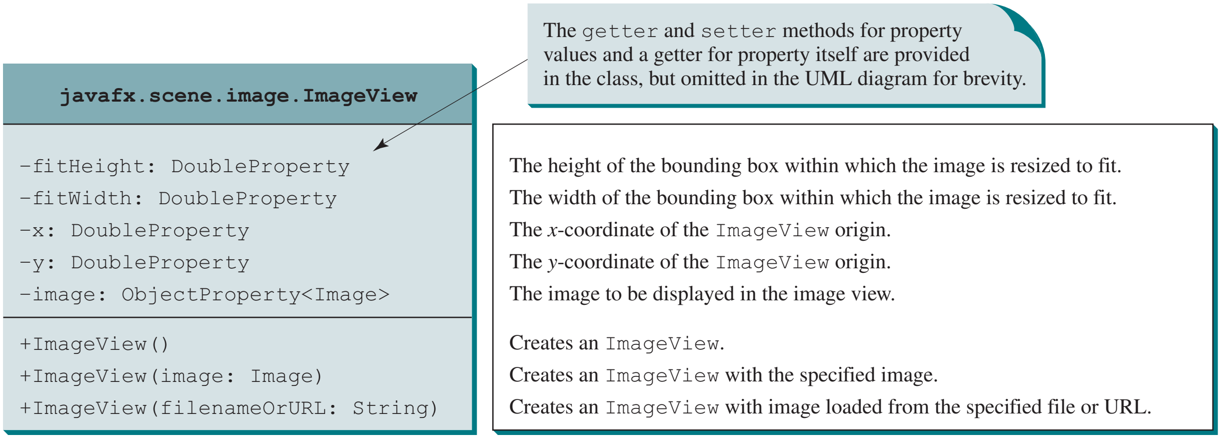

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
javafx.scene.image.ImageView
-fitHeight: DoubleProperty
-fitWidth: DoubleProperty
-x: DoubleProperty
-y: DoubleProperty
-image: ObjectProperty<Image>
+ImageView()
+ImageView(image: Image)
+ImageView(filenameOrURL: String)
The height of the bounding box within which the image is resized to fit.
The width of the bounding box within which the image is resized to fit.
The x-coordinate of the ImageView origin.
The y-coordinate of the ImageView origin.
The image to be displayed in the image view.
Creates an ImageView.
Creates an ImageView with the specified image.
Creates an ImageView with image loaded from the specified file or URL.

FIGURE 14.13 `ImageView` is a node for displaying an image.

Listing 14.9 displays an image in three image views, as shown in Figure 14.14.

LISTING 14.9 ShowImage.java

```
 1  import javafx.application.Application;
 2  import javafx.scene.Scene;
 3  import javafx.scene.layout.HBox;
 4  import javafx.scene.layout.Pane;
 5  import javafx.geometry.Insets;
 6  import javafx.stage.Stage;
 7  import javafx.scene.image.Image;
 8  import javafx.scene.image.ImageView;
 9
10  public class ShowImage extends Application {
11    @Override // Override the start method in the Application class
12    public void start(Stage primaryStage) {
13      // Create a pane to hold the image views
14      Pane pane = new HBox(10);
15      pane.setPadding(new Insets(5, 5, 5, 5));
16      Image image = new Image("image/us.gif");
17      pane.getChildren().add(new ImageView(image));
18
19      ImageView imageView2 = new ImageView(image);
20      imageView2.setFitHeight(100);
21      imageView2.setFitWidth(100);
22      pane.getChildren().add(imageView2);
23
24      ImageView imageView3 = new ImageView(image);
25      imageView3.setRotate(90);
26      pane.getChildren().add(imageView3);
27
28      // Create a scene and place it in the stage
29      Scene scene = new Scene(pane);
30      primaryStage.setTitle("ShowImage"); // Set the stage title
31      primaryStage.setScene(scene); // Place the scene in the stage
32      primaryStage.show(); // Display the stage
33    }
34  }
```

create an **HBox**
create an image
add an image view to pane
create an image view
set image view properties
add an image to pane
create an image view
rotate an image view
add an image to pane
main method omitted

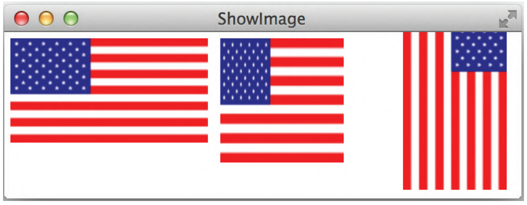
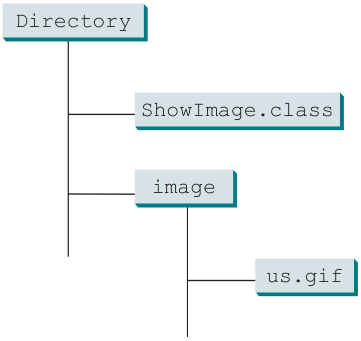


FIGURE 14.14 An image is displayed in three image views placed in a pane. *Source*: booka/Fotolia.

The program creates an **HBox** (line 14). An **HBox** is a pane that places all nodes horizontally in one row. The program creates an **Image**, then an **ImageView** for displaying the image, and places the **ImageView** in the **HBox** (line 17).

The program creates the second **ImageView** (line 19), sets its **fitHeight** and **fitWidth** properties (lines 20 and 21), and places the **ImageView** into the **HBox** (line 22). The program creates the third **ImageView** (line 24), rotates it 90 degrees (line 25), and places it into the **HBox** (line 26). The **setRotate** method is defined in the **Node** class and can be used for any node. Note an **Image** object can be shared by multiple nodes. In this case, it is shared by three **ImageView**. However, a node such as **ImageView** cannot be shared. You cannot place an **ImageView** multiple times into a pane or scene.

Note you must place the image file in the same directory as the class file, as shown in the following figure.



If you use the URL to locate the image file, the URL protocol **`http://`** must be present. Therefore, the following code is wrong:

```
new Image("liveexample.pearsoncmg.com/book/image/us.gif");
```

It must be replaced by

```
new Image("http://liveexample.pearsoncmg.com/book/image/us.gif");
```

Check Point

14.9.1 How do you create an **`Image`** from a URL or a filename?

14.9.2 How do you create an **`ImageView`** from an **`Image`** or directly from a file or a URL?

14.9.3 Can you set an **`Image`** to multiple **`ImageViews`**? Can you display the same **`ImageView`** multiple times?

14.10 Layout Panes and Groups

Key Point

JavaFX provides many types of panes for automatically laying out nodes in a desired location and size.

VideoNote
Use layout panes

Panes and groups are the containers for holding nodes. The **`Group`** class is often used to group nodes and to perform transformation and scale as a group. Panes and UI control objects are resizable, but group, shape, and text objects are not resizable. JavaFX provides many types of panes for organizing nodes in a container, as shown in Table 14.1. You have used the layout panes **`Pane`**, **`StackPane`**, and **`HBox`** in the preceding sections for containing nodes. This section introduces the panes in more details.

TABLE 14.1 Panes for Containing and Organizing Nodes

Class	*Description*
`Pane`	Base class for layout panes. It contains the **`getChildren()`** method for returning a list of nodes in the pane.
`StackPane`	Places the nodes on top of each other in the center of the pane.
`FlowPane`	Places the nodes row-by-row horizontally or column-by-column vertically.
`GridPane`	Places the nodes in the cells in a two-dimensional grid.
`BorderPane`	Places the nodes in the top, right, bottom, left, and center regions.
`HBox`	Places the nodes in a single row.
`VBox`	Places the nodes in a single column.

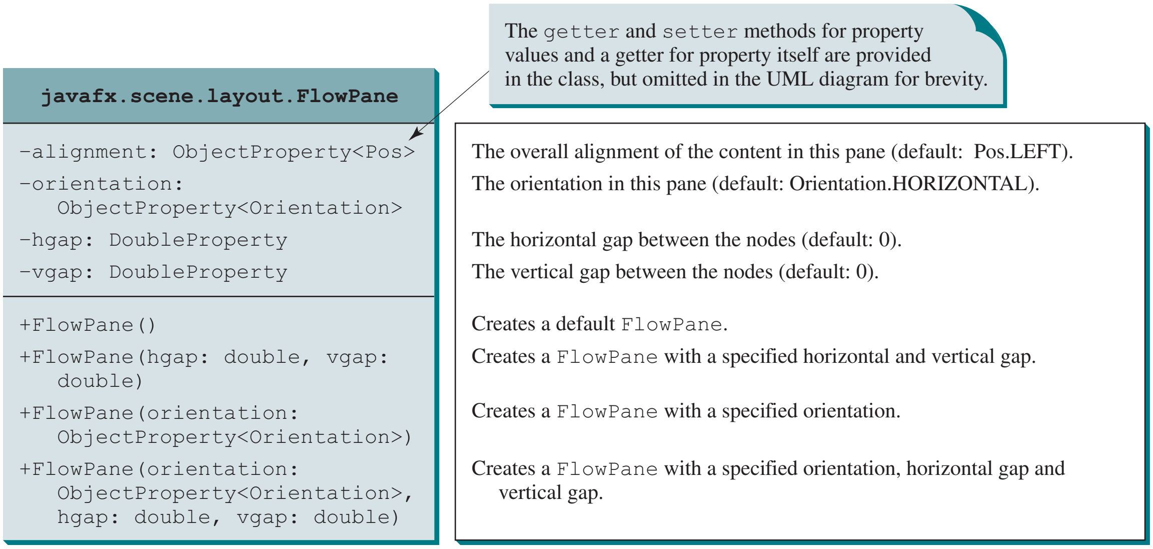
You have used the **`Pane`** in Listing 14.4, ShowCircle.java. A **`Pane`** is usually used as a canvas for displaying shapes. **`Pane`** is the base class for all specialized panes. You have used a specialized pane **`StackPane`** in Listing 14.3, ButtonInPane.java. Nodes are placed in the center of a **`StackPane`**. Each pane contains a list for holding nodes in the pane. This list is an instance of **`ObservableList`**, which can be obtained using pane's **`getChildren()`** method. You can use **`add(node)`** to add an element to the list and **`addAll(node1, node2, ...)`** to add a variable number of nodes.

`ObservableList`
`getChildren()`

14.10.1 FlowPane

`FlowPane` arranges the nodes in the pane horizontally from left to right, or vertically from top to bottom, in the order in which they were added. When one row or one column is filled, a new row or column is started. You can specify the way the nodes are placed horizontally or vertically using one of two constants: **`Orientation.HORIZONTAL`** or **`Orientation.VERTICAL`**. You can also specify the gap between the nodes in pixels. The class diagram for **`FlowPane`** is shown in Figure 14.15.

Data fields **`alignment`**, **`orientation`**, **`hgap`**, and **`vgap`** are binding properties. Recall that each binding property in JavaFX has a getter method (e.g., **`getHgap()`**) that returns its value, a setter method (e.g., **`setHGap(double)`**) for setting a value, and a getter method that returns the property itself (e.g., **`hgapProperty()`**). For a data field of **`ObjectProperty<T>`** type, the value getter method returns a value of type **`T`**, and the property getter method returns a property value of type **`ObjectProperty<T>`**.



The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.FlowPane	
-alignment: ObjectProperty<Pos>	The overall alignment of the content in this pane (default: Pos.LEFT).
-orientation: ObjectProperty<Orientation>	The orientation in this pane (default: Orientation.HORIZONTAL).
-hgap: DoubleProperty	The horizontal gap between the nodes (default: 0).
-vgap: DoubleProperty	The vertical gap between the nodes (default: 0).
+FlowPane()	Creates a default `FlowPane`.
+FlowPane(hgap: double, vgap: double)	Creates a `FlowPane` with a specified horizontal and vertical gap.
+FlowPane(orientation: ObjectProperty<Orientation>)	Creates a `FlowPane` with a specified orientation.
+FlowPane(orientation: ObjectProperty<Orientation>, hgap: double, vgap: double)	Creates a `FlowPane` with a specified orientation, horizontal gap and vertical gap.

FIGURE 14.15 **`FlowPane`** lays out nodes row-by-row horizontally or column-by-column vertically.

Listing 14.10 gives a program that demonstrates **`FlowPane`**. The program adds labels and text fields to a **`FlowPane`**, as shown in Figure 14.16.

LISTING 14.10 ShowFlowPane.java

```java
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;
```

extend `Application`

create `FlowPane`

add `UI` controls to pane

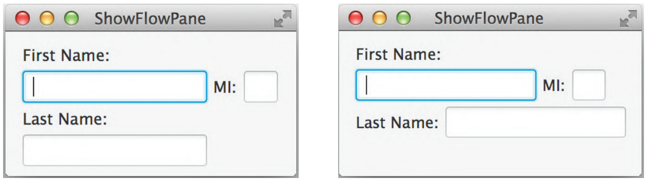
add pane to scene

place scene to stage

display stage

main method omitted

```
 8
 9 public class ShowFlowPane extends Application {
10   @Override // Override the start method in the Application class
11   public void start(Stage primaryStage) {
12     // Create a pane and set its properties
13     FlowPane pane = new FlowPane();
14     pane.setPadding(new Insets(11, 12, 13, 14));
15     pane.setHgap(5);
16     pane.setVgap(5);
17
18     // Place nodes in the pane
19     pane.getChildren().addAll(new Label("First Name:"),
20       new TextField(), new Label("MI:"));
21     TextField tfMi = new TextField();
22     tfMi.setPrefColumnCount(1);
23     pane.getChildren().addAll(tfMi, new Label("Last Name:"),
24       new TextField());
25
26     // Create a scene and place it in the stage
27     Scene scene = new Scene(pane, 200, 250);
28     primaryStage.setTitle("ShowFlowPane"); // Set the stage title
29     primaryStage.setScene(scene); // Place the scene in the stage
30     primaryStage.show(); // Display the stage
31   }
32 }
```


ShowFlowPane
First Name:
MI:
Last Name:
ShowFlowPane
First Name:
MI:
Last Name:

(a) (b)

FIGURE 14.16 The nodes fill in the rows in the `FlowPane` one after another.

The program creates a `FlowPane` (line 13) and sets its `padding` property with an `Insets` object (line 14). An `Insets` object specifies the size of the border of a pane. The constructor `Insets(11, 12, 13, 14)` creates an `Insets` with the border sizes for top (11), right (12), bottom (13), and left (14) in pixels, as shown in Figure 14.17. You can also use the constructor `Insets(value)` to create an `Insets` with the same value for all four sides. The `hGap` and `vGap` properties are in lines 15 and 16 to specify the horizontal gap and vertical gap, respectively, between two nodes in the pane, as shown in Figure 14.17.

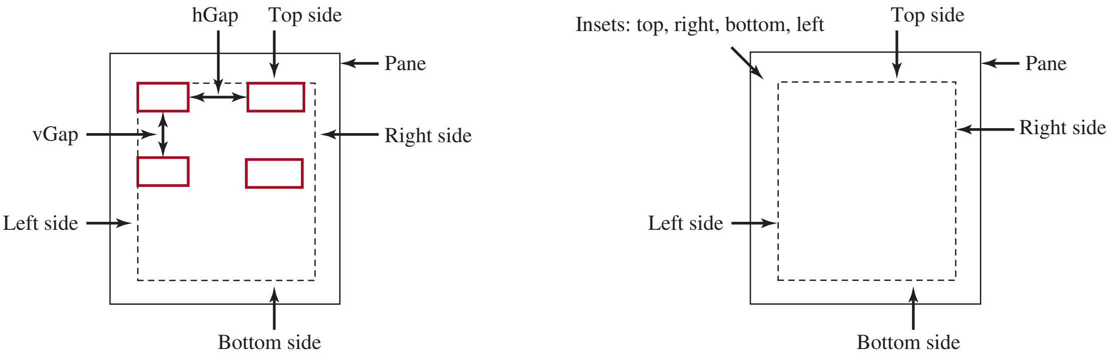

hGap
Top side
Pane
vGap
Right side
Left side
Bottom side
Insets: top, right, bottom, left
Top side
Pane
Right side
Left side
Bottom side

FIGURE 14.17 You can specify `hGap` and `vGap` between the nodes in a `FlowLPane`.

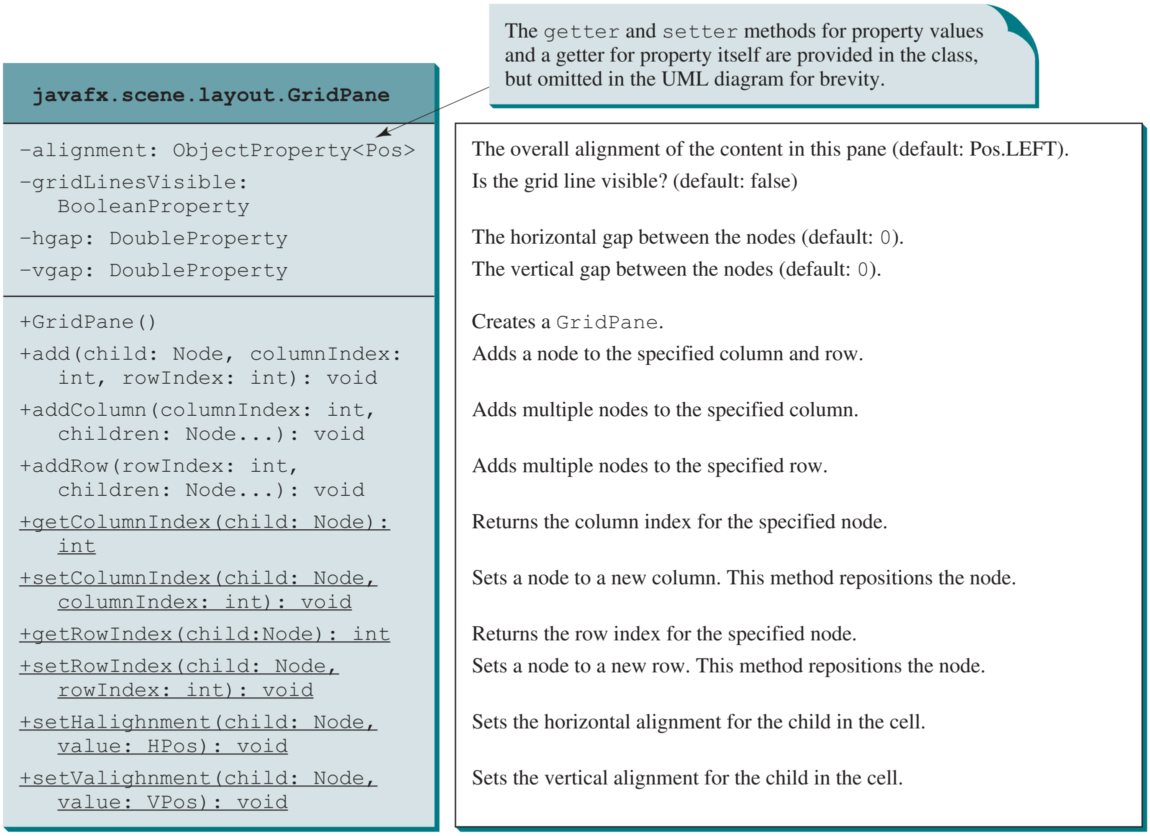
Each `FlowPane` contains an object of `ObservableList` for holding the nodes. This list can be obtained using the `getChildren()` method (line 19). To add a node into a `FlowPane` is to add it to this list using the `add(node)` or `addAll(node1, node2, ...)` method. You can also remove a node from the list using the `remove(node)` method, or use the `removeAll()` method to remove all nodes from the pane. The program adds the labels and text fields into the pane (lines 19–24). Invoking `tfMi.setPrefColumnCount(1)` sets the preferred column count to `1` for the MI text field (line 22). The program declares an explicit reference `tfMi` for a `TextField` object for MI. The explicit reference is necessary because we need to reference the object directly to set its `prefColumnCount` property.

The program adds the pane to the scene (line 27), sets the scene in the stage (line 29), and displays the stage (line 30). Note if you resize the window, the nodes are automatically rearranged to fit in the pane. In Figure 14.16a, the first row has three nodes, but in Figure 14.16b, the first row has four nodes because the width has been increased.

Suppose you wish to add the object `tfMi` to a pane 10 times; will 10 text fields appear in the pane? No, a node such as a text field can be added to only one pane and once. Adding a node to a pane multiple times or to different panes will cause a runtime error.

14.10.2 GridPane

A `GridPane` arranges nodes in a grid (matrix) formation. The nodes are placed in the specified column and row indices. The class diagram for `GridPane` is shown in Figure 14.18.



The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.GridPane	
-alignment: ObjectProperty<Pos>	The overall alignment of the content in this pane (default: Pos.LEFT).
-gridLinesVisible: BooleanProperty	Is the grid line visible? (default: false)
-hgap: DoubleProperty	The horizontal gap between the nodes (default: 0).
-vgap: DoubleProperty	The vertical gap between the nodes (default: 0).
+GridPane()	Creates a GridPane.
+add(child: Node, columnIndex: int, rowIndex: int): void	Adds a node to the specified column and row.
+addColumn(columnIndex: int, children: Node...): void	Adds multiple nodes to the specified column.
+addRow(rowIndex: int, children: Node...): void	Adds multiple nodes to the specified row.
+getColumnIndex(child: Node): int	Returns the column index for the specified node.
+setColumnIndex(child: Node, columnIndex: int): void	Sets a node to a new column. This method repositions the node.
+getRowIndex(child:Node): int	Returns the row index for the specified node.
+setRowIndex(child: Node, rowIndex: int): void	Sets a node to a new row. This method repositions the node.
+setHalighnment(child: Node, value: HPos): void	Sets the horizontal alignment for the child in the cell.
+setValighnment(child: Node, value: VPos): void	Sets the vertical alignment for the child in the cell.

FIGURE 14.18 `GridPane` lays out nodes in the specified cell in a grid.

Listing 14.11 gives a program that demonstrates `GridPane`. The program is similar to the one in Listing 14.10, except that it adds three labels and three text fields, and a button to the specified location in a grid, as shown in Figure 14.19.

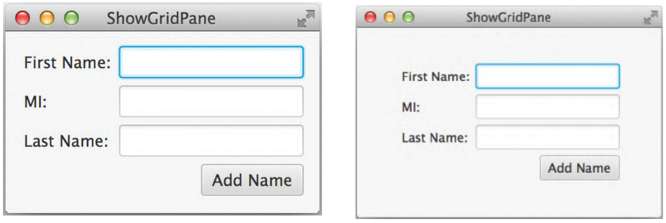


FIGURE 14.19 The **`GridPane`** places the nodes in a grid with a specified column and row indices.

LISTING 14.11 ShowGridPane.java

```
 1  import javafx.application.Application;
 2  import javafx.geometry.HPos;
 3  import javafx.geometry.Insets;
 4  import javafx.geometry.Pos;
 5  import javafx.scene.Scene;
 6  import javafx.scene.control.Button;
 7  import javafx.scene.control.Label;
 8  import javafx.scene.control.TextField;
 9  import javafx.scene.layout.GridPane;
10  import javafx.stage.Stage;
11
12  public class ShowGridPane extends Application {
13    @Override // Override the start method in the Application class
14    public void start(Stage primaryStage) {
15      // Create a pane and set its properties
16      GridPane pane = new GridPane();
17      pane.setAlignment(Pos.CENTER);
18      pane.setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
19      pane.setHgap(5.5);
20      pane.setVgap(5.5);
21
22      // Place nodes in the pane
23      pane.add(new Label("First Name:"), 0, 0);
24      pane.add(new TextField(), 1, 0);
25      pane.add(new Label("MI:"), 0, 1);
26      pane.add(new TextField(), 1, 1);
27      pane.add(new Label("Last Name:"), 0, 2);
28      pane.add(new TextField(), 1, 2);
29      Button btAdd = new Button("Add Name");
30      pane.add(btAdd, 1, 3);
31      GridPane.setHalignment(btAdd, HPos.RIGHT);
32
33      // Create a scene and place it in the stage
34      Scene scene = new Scene(pane);
35      primaryStage.setTitle("ShowGridPane"); // Set the stage title
36      primaryStage.setScene(scene); // Place the scene in the stage
37      primaryStage.show(); // Display the stage
38    }
39  }
```

create a grid pane
set properties
add label
add text field
add button
align button right
create a scene
display stage
main method omitted

The program creates a **`GridPane`** (line 16) and sets its properties (line 17–20). The alignment is set to the center position (line 17), which causes the nodes to be placed in the center of the grid pane. If you resize the window, you will see the nodes remained in the center of the grid pane.

The program adds the label in column `0` and row `0` (line 23). The column and row index starts from `0`. The `add` method places a node in the specified column and row. Not every cell in the grid needs to be filled. A button is placed in column 1 and row 3 (line 30), but there are no nodes placed in column 0 and row 3. To remove a node from a `GridPane`, use `pane.getChildren().remove(node)`. To remove all nodes, use `pane.getChildren().removeAll()`.

remove nodes

The program invokes the static `setHalignment` method to align the button right in the cell (line 31).

Note the scene size is not set (line 34). In this case, the scene size is automatically computed according to the sizes of the nodes placed inside the scene.

By default, the grid pane will resize rows and columns to the preferred sizes of its contents, even if the grid pane is resized larger than its preferred size. You may purposely set a large value for the preferred width and height of its contents by invoking the `setPrefWidth` and `setPrefHeight` methods, so the contents will be automatically stretched to fill in the grid pane when the grid pane is enlarged (see Programming Exercise 14.8).

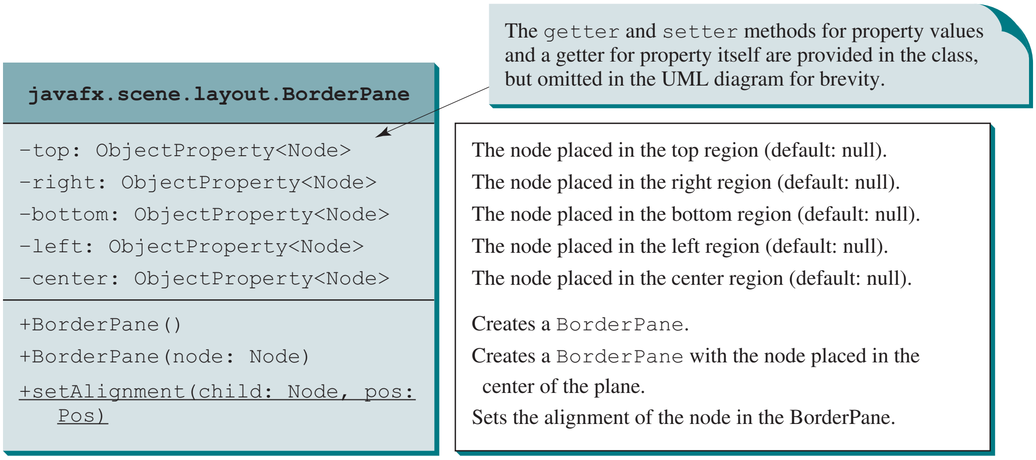
remove nodes

`setPrefWidth`

`setPrefHeight`

14.10.3 BorderPane

A `BorderPane` can place nodes in five regions: top, bottom, left, right, and center, using the `setTop(node)`, `setBottom(node)`, `setLeft(node)`, `setRight(node)`, and `setCenter(node)` methods. The class diagram for `BorderPane` is shown in Figure 14.20.



The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.BorderPane	
-top: ObjectProperty<Node>	The node placed in the top region (default: null).
-right: ObjectProperty<Node>	The node placed in the right region (default: null).
-bottom: ObjectProperty<Node>	The node placed in the bottom region (default: null).
-left: ObjectProperty<Node>	The node placed in the left region (default: null).
-center: ObjectProperty<Node>	The node placed in the center region (default: null).
+BorderPane()	Creates a BorderPane.
+BorderPane(node: Node)	Creates a BorderPane with the node placed in the center of the plane.
+setAlignment(child: Node, pos: Pos)	Sets the alignment of the node in the BorderPane.

FIGURE 14.20 `BorderPane` places the nodes in top, bottom, left, right, and center regions.

Listing 14.12 gives a program that demonstrates `BorderPane`. The program places five buttons in the five regions of the pane, as shown in Figure 14.21.

LISTING 14.12 ShowBorderPane.java

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class ShowBorderPane extends Application {
  @Override // Override the start method in the Application class
```

create a border pane
add to top
add to right
add to bottom
add to left
add to center
define a custom pane
add a label to pane
set style
set padding
main method omitted

```
11     public void start(Stage primaryStage) {
12       // Create a border pane
13       BorderPane pane = new BorderPane();
14 
15       // Place nodes in the pane
16       pane.setTop(new CustomPane("Top"));
17       pane.setRight(new CustomPane("Right"));
18       pane.setBottom(new CustomPane("Bottom"));
19       pane.setLeft(new CustomPane("Left"));
20       pane.setCenter(new CustomPane("Center"));
21 
22       // Create a scene and place it in the stage
23       Scene scene = new Scene(pane);
24       primaryStage.setTitle("ShowBorderPane"); // Set the stage title
25       primaryStage.setScene(scene); // Place the scene in the stage
26       primaryStage.show(); // Display the stage
27     }
28   }
29 
30   // Define a custom pane to hold a label in the center of the pane
31   class CustomPane extends StackPane {
32     public CustomPane(String title) {
33       getChildren().add(new Label(title));
34       setStyle("-fx-border-color: red");
35       setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
36     }
37   }
```

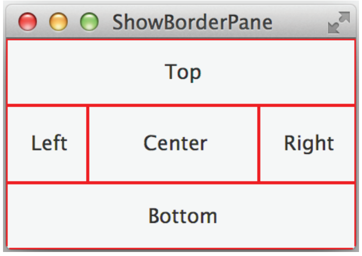


FIGURE 14.21 The `BorderPane` places the nodes in five regions of the pane.

The program defines `CustomPane` that extends `StackPane` (line 31). The constructor of `CustomPane` adds a label with the specified title (line 33), sets a style for the border color, and sets a padding using insets (line 35).

The program creates a `BorderPane` (line 13) and places five instances of `CustomPane` into five regions of the border pane (lines 16–20). Note a pane is a node. Therefore, a pane can be added into another pane. To remove a node from the top region, invoke `setTop(null)`. If a region is not occupied, no space will be allocated for this region.

14.10.4 HBox and VBox

An `HBox` lays out its children in a single horizontal row. A `VBox` lays out its children in a single vertical column. Recall that a `FlowPane` can lay out its children in multiple rows or multiple columns, but an `HBox` or a `VBox` can lay out children only in one row or one column. The class diagrams for `HBox` and `VBox` are shown in Figures 14.22 and 14.23.

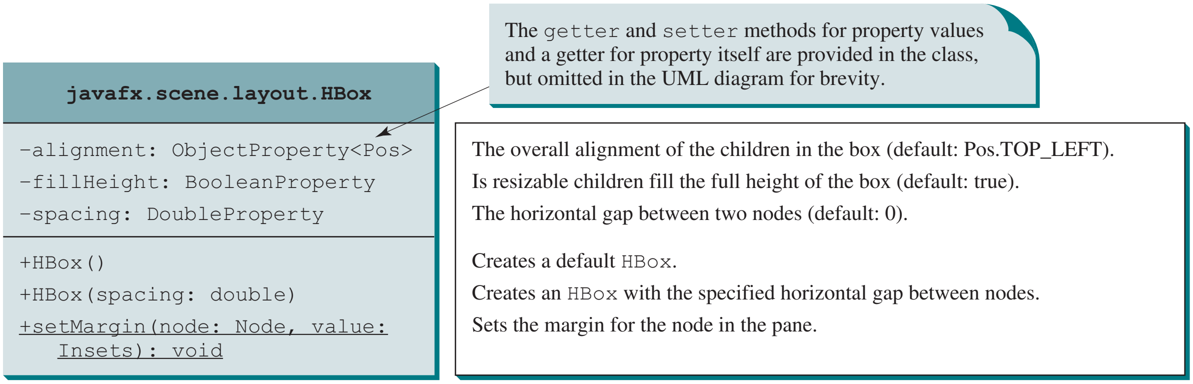

javafx.scene.layout.HBox
-alignment: ObjectProperty<Pos>
-fillHeight: BooleanProperty
-spacing: DoubleProperty
+HBox()
+HBox(spacing: double)
+setMargin(node: Node, value: Insets): void
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
The overall alignment of the children in the box (default: Pos.TOP_LEFT).
Is resizable children fill the full height of the box (default: true).
The horizontal gap between two nodes (default: 0).
Creates a default HBox.
Creates an HBox with the specified horizontal gap between nodes.
Sets the margin for the node in the pane.

FIGURE 14.22 **HBox** places the nodes in one row.

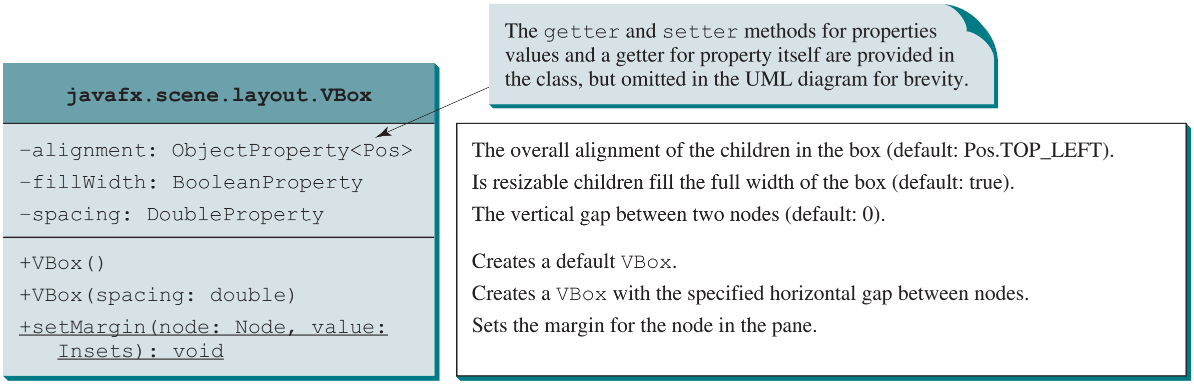

javafx.scene.layout.VBox
-alignment: ObjectProperty<Pos>
-fillWidth: BooleanProperty
-spacing: DoubleProperty
+VBox()
+VBox(spacing: double)
+setMargin(node: Node, value: Insets): void
The getter and setter methods for properties values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
The overall alignment of the children in the box (default: Pos.TOP_LEFT).
Is resizable children fill the full width of the box (default: true).
The vertical gap between two nodes (default: 0).
Creates a default VBox.
Creates a VBox with the specified horizontal gap between nodes.
Sets the margin for the node in the pane.

FIGURE 14.23 **VBox** places the nodes in one column.

Listing 14.13 gives a program that demonstrates **HBox** and **VBox**. The program places two buttons and an image view in an **HBox** and five labels in a **VBox**, as shown in Figure 14.24.

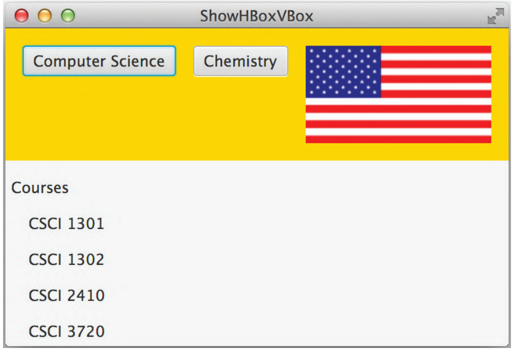
LISTING 14.13 ShowHBoxVBox.java

```java
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;

public class ShowHBoxVBox extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a border pane
    BorderPane pane = new BorderPane();

```

create a border pane

add an `HBox` to top
add a `VBox` to left
create a scene
display stage
`getHBox`
add buttons to `HBox`
return an `HBox`
`getVBox`
add a label
set margin
add a label
return `vBox`
main method omitted

```
19     // Place nodes in the pane
20     pane.setTop(getHBox());
21     pane.setLeft(getVBox());
22 
23     // Create a scene and place it in the stage
24     Scene scene = new Scene(pane);
25     primaryStage.setTitle("ShowHBoxVBox"); // Set the stage title
26     primaryStage.setScene(scene); // Place the scene in the stage
27     primaryStage.show(); // Display the stage
28   }
29 
30   private HBox getHBox() {
31     HBox hBox = new HBox(15);
32     hBox.setPadding(new Insets(15, 15, 15, 15));
33     hBox.setStyle("-fx-background-color: gold");
34     hBox.getChildren().add(new Button("Computer Science"));
35     hBox.getChildren().add(new Button("Chemistry"));
36     ImageView imageView = new ImageView(new Image("image/us.gif"));
37     hBox.getChildren().add(imageView);
38     return hBox;
39   }
40 
41   private VBox getVBox() {
42     VBox vBox = new VBox(15);
43     vBox.setPadding(new Insets(15, 5, 5, 5));
44     vBox.getChildren().add(new Label("Courses"));
45 
46     Label[] courses = {new Label("CSCI 1301"), new Label("CSCI 1302"),
47         new Label("CSCI 2410"), new Label("CSCI 3720")};
48 
49     for (Label course: courses) {
50       VBox.setMargin(course, new Insets(0, 0, 0, 15));
51       vBox.getChildren().add(course);
52     }
53 
54     return vBox;
55   }
56 }
```



Courses
CSCI 1301
CSCI 1302
CSCI 2410
CSCI 3720

FIGURE 14.24 The **HBox** places the nodes in one row, and the **VBox** places the nodes in one column. *Source*: booka/Fotolia.

The program defines the **getHBox()** method. This method returns an **HBox** that contains two buttons and an image view (lines 30–39). The background color of the **HBox** is set to gold using Java CSS (line 33). The program defines the **getVBox()** method. This method returns a **VBox** that contains five labels (lines 41–55). The first label is added to the **VBox** in line 44 and the other four are added in line 51. The `setMargin` method is used to set a node's margin when placed inside the `VBox` (line 50).

Check Point

14.10.1 How do you add a node to a `Pane`, `StackPane`, `FlowPane`, `GridPane`, `BorderPane`, `HBox`, and `VBox`? How do you remove a node from these panes?

14.10.2 How do you set the alignment to right for nodes in a `FlowPane`, `GridPane`, `HBox`, and `VBox`?

14.10.3 How do you set the horizontal gap and vertical gap between nodes in 8 pixels in a `FlowPane` and `GridPane` and set spacing in 8 pixels in an `HBox` and `VBox`?

14.10.4 How do you get the column and row index of a node in a `GridPane`? How do you reposition a node in a `GridPane`?

14.10.5 What are the differences between a `FlowPane` and an `HBox` or a `VBox`?

14.11 Shapes

Key Point

JavaFX provides many shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines.

VideoNote
Use shapes

`fill` property
`stroke` property
`strokeWidth` property

The `Shape` class is the abstract base class that defines the common properties for all shapes. Among them are the `fill`, `stroke`, and `strokeWidth` properties. The `fill` property specifies a color that fills the interior of a shape. The `stroke` property specifies a color that is used to draw the outline of a shape. The `strokeWidth` property specifies the width of the outline of a shape. This section introduces the classes `Text`, `Line`, `Rectangle`, `Circle`, `Ellipse`, `Arc`, `Polygon`, and `Polyline` for drawing texts and simple shapes. All these are subclasses of `Shape`, as shown in Figure 14.25.

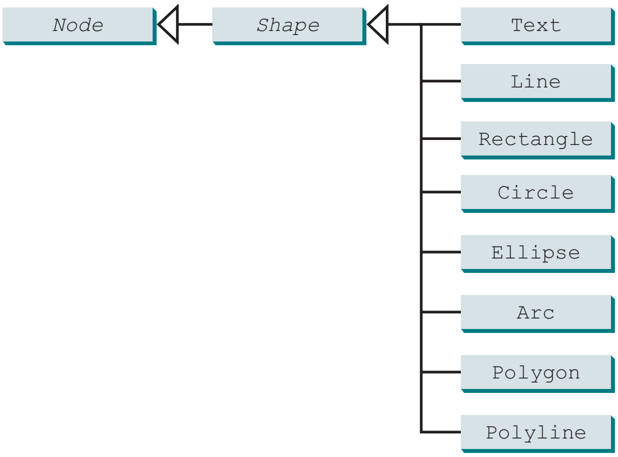
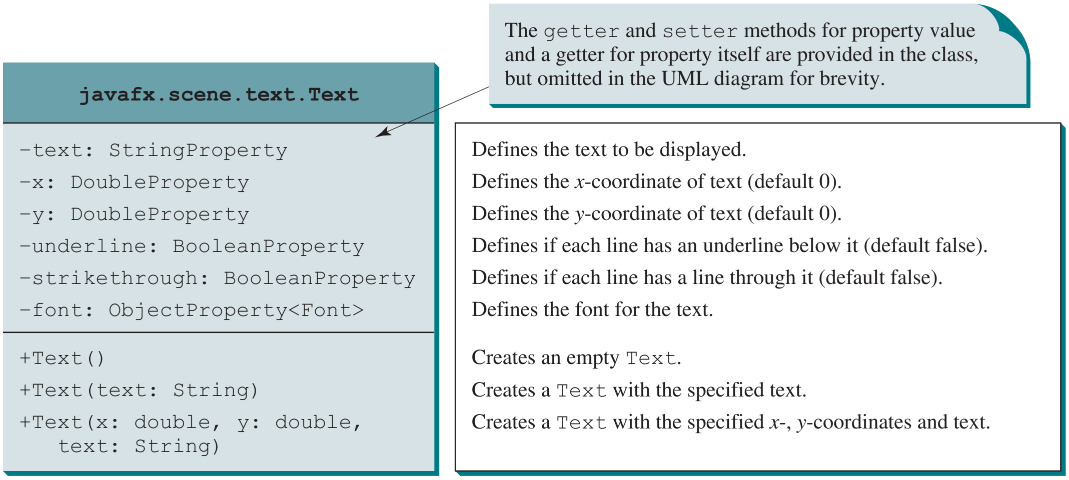


FIGURE 14.25 A shape is a node. The `Shape` class is the root of all shape classes.

14.11.1 Text

The `Text` class defines a node that displays a string at a starting point (`x`, `y`), as shown in Figure 14.27a. A `Text` object is usually placed in a pane. The pane's upper-left corner point is (`0`, `0`) and the bottom-right point is (`pane.getWidth()`, `pane.getHeight()`). A string may be displayed in multiple lines separated by `\n`. The UML diagram for the `Text` class is shown in Figure 14.26. Listing 14.14 gives an example that demonstrates text, as shown in Figure 14.27b.

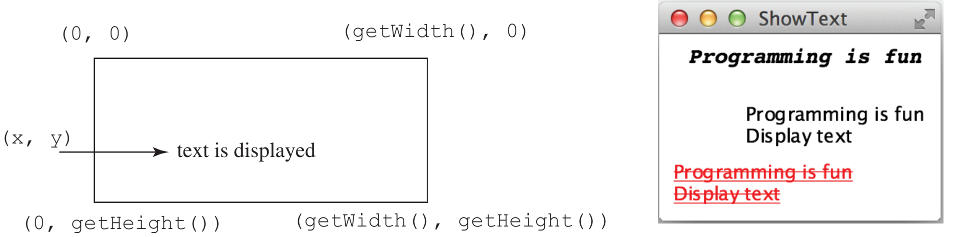


The `getter` and `setter` methods for property value and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.text.Text

`-text: StringProperty`	Defines the text to be displayed.
`-x: DoubleProperty`	Defines the *x*-coordinate of text (default 0).
`-y: DoubleProperty`	Defines the *y*-coordinate of text (default 0).
`-underline: BooleanProperty`	Defines if each line has an underline below it (default false).
`-strikethrough: BooleanProperty`	Defines if each line has a line through it (default false).
`-font: ObjectProperty<Font>`	Defines the font for the text.
`+Text()`	Creates an empty `Text`.
`+Text(text: String)`	Creates a `Text` with the specified text.
`+Text(x: double, y: double, text: String)`	Creates a `Text` with the specified *x*-, *y*-coordinates and text.

FIGURE 14.26 `Text` defines a node for displaying a text.



Programming is fun

Programming is fun
Display text

~~Programming is fun~~
~~Display text~~

(a) `Text`(*x*, *y*, text) (b) Three `Text` objects are displayed

FIGURE 14.27 A Text object is created to display a text.

LISTING 14.14 ShowText.java

create a pane

create a `Text`

set text font

add text to pane

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.geometry.Insets;
import javafx.stage.Stage;
import javafx.scene.text.Text;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.FontPosture;

public class ShowText extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane to hold the texts
    Pane pane = new Pane();
    pane.setPadding(new Insets(5, 5, 5, 5));
    Text text1 = new Text(20, 20, "Programming is fun");
    text1.setFont(Font.font("Courier", FontWeight.BOLD,
      FontPosture.ITALIC, 15));
    pane.getChildren().add(text1);
22
23      Text text2 = new Text(60, 60, "Programming is fun\nDisplay text");
24      pane.getChildren().add(text2);
25
26      Text text3 = new Text(10, 100, "Programming is fun\nDisplay text");
27      text3.setFill(Color.RED);
28      text3.setUnderline(true);
29      text3.setStrikethrough(true);
30      pane.getChildren().add(text3);
31
32      // Create a scene and place it in the stage
33      Scene scene = new Scene(pane);
34      primaryStage.setTitle("ShowText"); // Set the stage title
35      primaryStage.setScene(scene); // Place the scene in the stage
36      primaryStage.show(); // Display the stage
37    }
38  }
```

create a two-line Text
add text to pane
create a Text
set text color
set underline
set strike line
add text to pane
main method omitted

The program creates a **Text** (line 18), sets its font (line 19), and places it to the pane (line 21). The program creates another **Text** with multiple lines (line 23) and places it to the pane (line 24). The program creates the third **Text** (line 26), sets its color (line 27), sets an underline and a strike through line (lines 28 and 29), and places it to the pane (line 30).

14.11.2 Line

A line connects two points with four parameters **startX**, **startY**, **endX**, and **endY**, as shown in Figure 14.29a. The **Line** class defines a line. The UML diagram for the **Line** class is shown in Figure 14.28. Listing 14.15 gives an example that demonstrates the line shape, as shown in Figure 14.29b.

LISTING 14.15 ShowLine.java

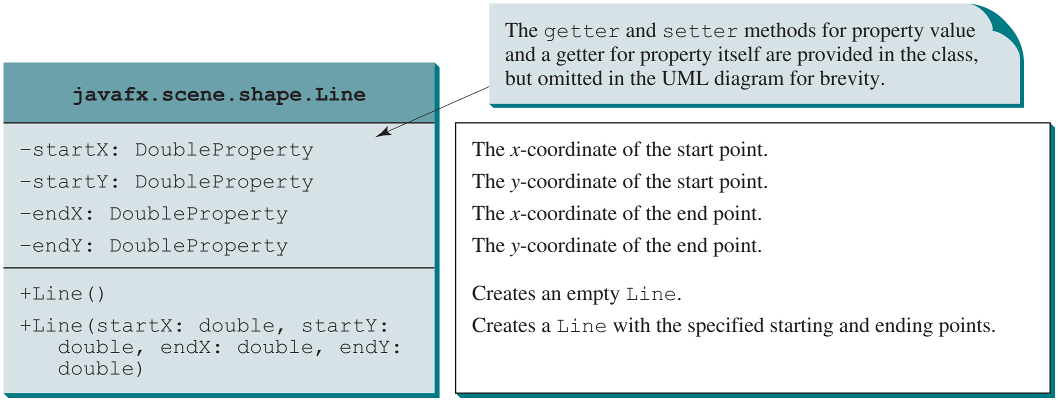
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Line;

public class ShowLine extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place it in the stage
    Scene scene = new Scene(new LinePane(), 200, 200);
    primaryStage.setTitle("ShowLine"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}

class LinePane extends Pane {
  public LinePane() {
    Line line1 = new Line(10, 10, 10, 10);
    line1.endXProperty().bind(widthProperty().subtract(10));
    line1.endYProperty().bind(heightProperty().subtract(10));
    line1.setStrokeWidth(5);
    line1.setStroke(Color.GREEN);
    getChildren().add(line1);
```

create a pane in scene
main method omitted
define a custom pane
create a line
set stroke width
set stroke
add line to pane

create a line

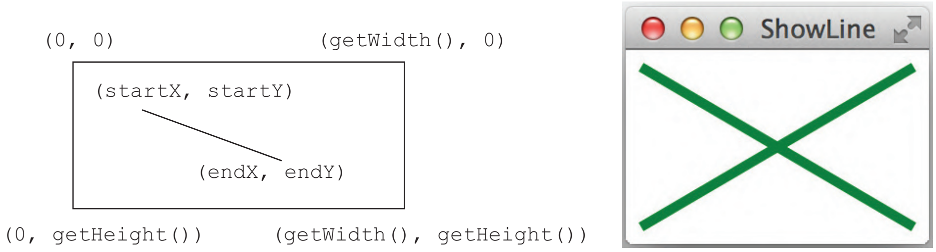
add line to pane

```
27
28       Line line2 = new Line(10, 10, 10, 10);
29       line2.startXProperty().bind(widthProperty().subtract(10));
30       line2.endYProperty().bind(heightProperty().subtract(10));
31       line2.setStrokeWidth(5);
32       line2.setStroke(Color.GREEN);
33       getChildren().add(line2);
34     }
35   }
```


The `getter` and `setter` methods for property value and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.shape.Line
-startX: DoubleProperty
-startY: DoubleProperty
-endX: DoubleProperty
-endY: DoubleProperty
+Line()
+Line(startX: double, startY: double, endX: double, endY: double)
The x-coordinate of the start point.
The y-coordinate of the start point.
The x-coordinate of the end point.
The y-coordinate of the end point.
Creates an empty Line.
Creates a Line with the specified starting and ending points.

FIGURE 14.28 The **`Line`** class defines a line.


(0, 0)
(getWidth(), 0)
(startX, startY)
(endX, endY)
(0, getHeight())
(getWidth(), getHeight())
ShowLine

(a) `Line(startX, startY, endX, endY)`

(b) Two lines are displayed across the pane.

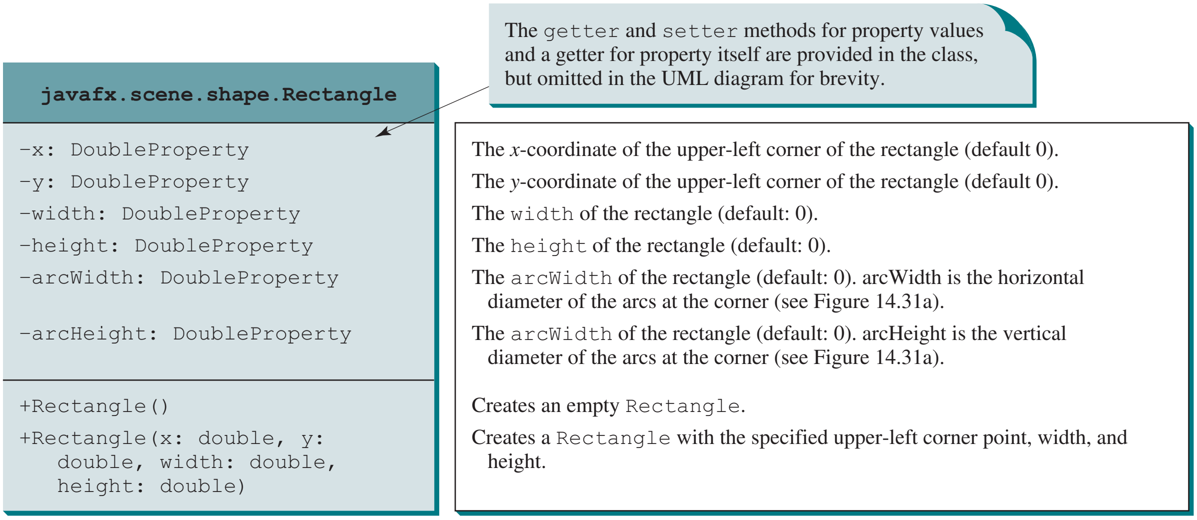
FIGURE 14.29 A **`Line`** object is created to display a line.

The program defines a custom pane class named **`LinePane`** (line 19). The custom pane class creates two lines and binds the starting and ending points of the line with the width and height of the pane (lines 22, 23, 29, and 30) so the two points of the lines are changed as the pane is resized.

14.11.3 Rectangle

A rectangle is defined by the parameters **`x`**, **`y`**, **`width`**, **`height`**, **`arcWidth`**, and **`arcHeight`**, as shown in Figure 14.31a. The rectangle's upper-left corner point is at (**`x`**, **`y`**), parameter **`aw`** (**`arcWidth`**) is the horizontal diameter of the arcs at the corner, and **`ah`** (**`arcHeight`**) is the vertical diameter of the arcs at the corner.

The **`Rectangle`** class defines a rectangle. The UML diagram for the **`Rectangle`** class is shown in Figure 14.30. Listing 14.16 gives an example that demonstrates rectangles, as shown in Figure 14.31b.


javafx.scene.shape.Rectangle
-x: DoubleProperty
-y: DoubleProperty
-width: DoubleProperty
-height: DoubleProperty
-arcWidth: DoubleProperty
-arcHeight: DoubleProperty
+Rectangle()
+Rectangle(x: double, y: double, width: double, height: double)

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The *x*-coordinate of the upper-left corner of the rectangle (default 0).

The *y*-coordinate of the upper-left corner of the rectangle (default 0).

The `width` of the rectangle (default: 0).

The `height` of the rectangle (default: 0).

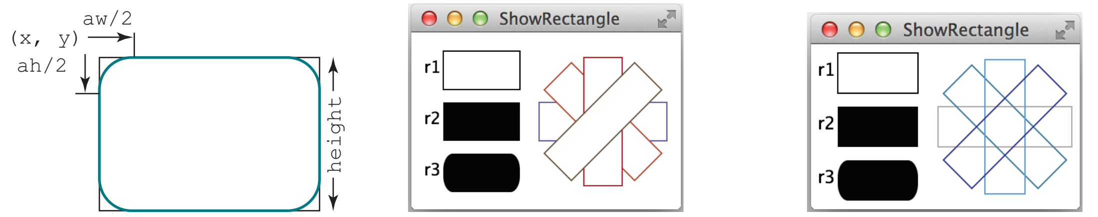
The `arcWidth` of the rectangle (default: 0). arcWidth is the horizontal diameter of the arcs at the corner (see Figure 14.31a).

The `arcWidth` of the rectangle (default: 0). arcHeight is the vertical diameter of the arcs at the corner (see Figure 14.31a).

Creates an empty `Rectangle`.

Creates a `Rectangle` with the specified upper-left corner point, width, and height.

FIGURE 14.30 `Rectangle` defines a rectangle.


aw/2
(x, y)
ah/2
height
width
ShowRectangle
r1
r2
r3
ShowRectangle
r1
r2
r3

(a) `Rectangle(x, y, w, h)` (b) Multiple rectangles are displayed. (c) Transparent rectangles are displayed.

FIGURE 14.31 A `Rectangle` object is created to display a rectangle.

LISTING 14.16 ShowRectangle.java

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.text.Text;
import javafx.scene.shape.Rectangle;

public class ShowRectangle extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create rectangles
    Rectangle r1 = new Rectangle(25, 10, 60, 30);
    r1.setStroke(Color.BLACK);
    r1.setFill(Color.WHITE);
    Rectangle r2 = new Rectangle(25, 50, 60, 30);
    Rectangle r3 = new Rectangle(25, 90, 60, 30);
    r3.setArcWidth(15);
    r3.setArcHeight(25);

```

create a rectangle `r1`

set `r1`'s properties

create rectangle `r2`

create rectangle `r3`

set `r3`'s arc width

set `r3`'s arc height

create a group
add nodes to group
create a rectangle
rotate a rectangle
add rectangle to group
main method omitted

```
22     // Create a group and add nodes to the group
23     Group group = new Group();
24     group.getChildren().addAll(new Text(10, 27, "r1"), r1,
25       new Text(10, 67, "r2"), r2, new Text(10, 107, "r3"), r3);
26
27     for (int i = 0; i < 4; i++) {
28       Rectangle r = new Rectangle(100, 50, 100, 30);
29       r.setRotate(i * 360 / 8);
30       r.setStroke(Color.color(Math.random(), Math.random(),
31         Math.random()));
32       r.setFill(Color.WHITE);
33       group.getChildren().add(r);
34     }
35
36     // Create a scene and place it in the stage
37     Scene scene = new Scene(new BorderPane(group),250, 150);
38     primaryStage.setTitle("ShowRectangle"); // Set the stage title
39     primaryStage.setScene(scene); // Place the scene in the stage
40     primaryStage.show(); // Display the stage
41   }
42 }
```

The program creates multiple rectangles. By default, the fill color is black. Thus, a rectangle is filled with black color. The stroke color is white by default. Line 15 sets stroke color of rectangle `r1` to black. The program creates rectangle `r3` (line 18) and sets its arc width and arc height (lines 19 and 20). Thus, `r3` is displayed as a rounded rectangle.

The program creates a `Group` to hold the nodes (lines 23–25). The program repeatedly creates a rectangle (line 28), rotates it (line 29), sets a random stroke color (lines 30 and 31), its fill color to white (line 32), and adds the rectangle to the group (line 33).

If line 32 is replaced by the following line:

```
r.setFill(null);
```

the rectangle is not filled with a color. Thus, they are displayed as shown in Figure 14.31c.

To center the nodes in the window, the program creates a `BorderPane` with the group in the center of the pane (line 37). If line 23 is replaced by the following?

```
Pane group = new Pane();
```

the rectangle will not be centered in the window. Therefore, using `Group` along with the `BorderPane` displays the contents of the group in the center of the window. Another advantage of using group is you can apply transformation to all nodes in the group. For example, if you add the following two lines in line 35:

```
group.setScaleX(2);
group.setScaleY(2);
```

the sizes of the nodes in the group are doubled.

14.11.4 `Circle` and `Ellipse`

You have used circles in several examples early in this chapter. A circle is defined by its parameters `centerX`, `centerY`, and `radius`. The `Circle` class defines a circle. The UML diagram for the `Circle` class is shown in Figure 14.32.

An ellipse is defined by its parameters `centerX`, `centerY`, `radiusX`, and `radiusY`, as shown in Figure 14.34a. The `Ellipse` class defines an ellipse. The UML diagram for the `Ellipse` class is shown in Figure 14.33. Listing 14.17 gives an example that demonstrates ellipses, as shown in Figure 14.34b.

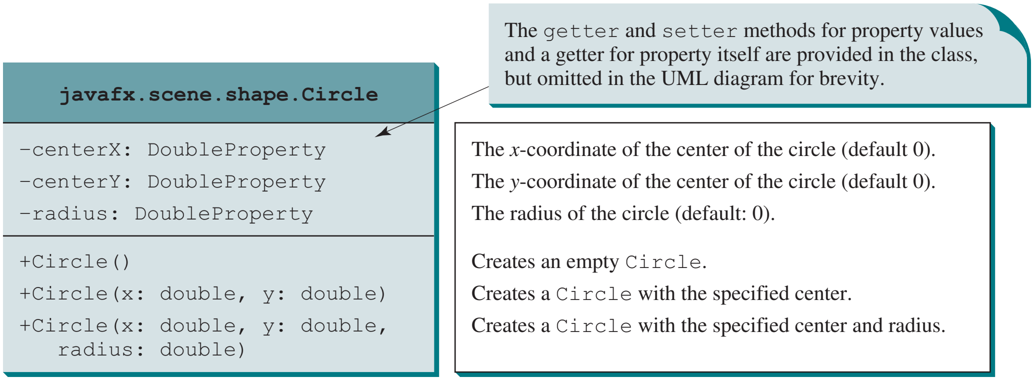

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
javafx.scene.shape.Circle
-centerX: DoubleProperty
-centerY: DoubleProperty
-radius: DoubleProperty
+Circle()
+Circle(x: double, y: double)
+Circle(x: double, y: double, radius: double)
The x-coordinate of the center of the circle (default 0).
The y-coordinate of the center of the circle (default 0).
The radius of the circle (default: 0).
Creates an empty Circle.
Creates a Circle with the specified center.
Creates a Circle with the specified center and radius.

FIGURE 14.32 The `Circle` class defines circles.

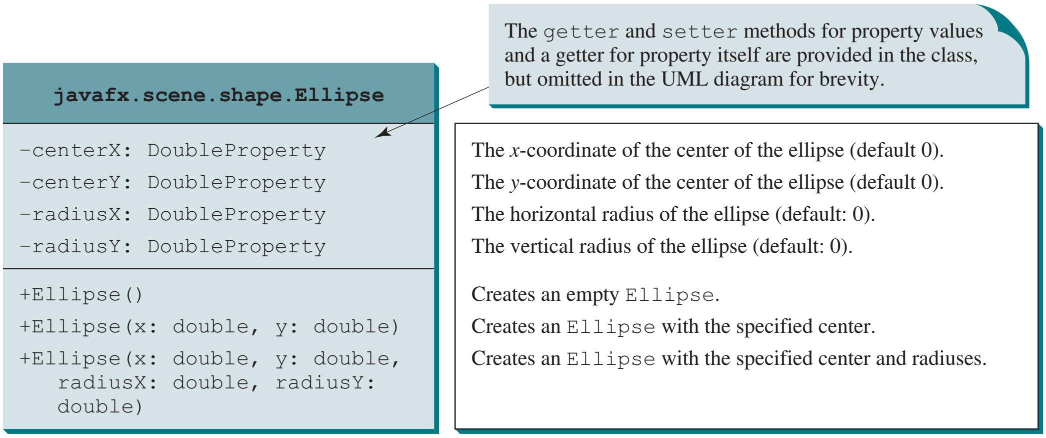
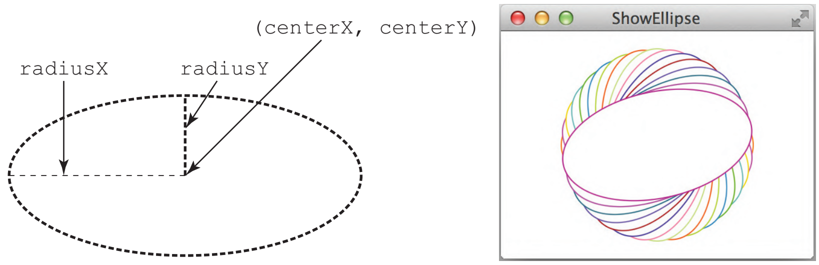

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
javafx.scene.shape.Ellipse
-centerX: DoubleProperty
-centerY: DoubleProperty
-radiusX: DoubleProperty
-radiusY: DoubleProperty
+Ellipse()
+Ellipse(x: double, y: double)
+Ellipse(x: double, y: double, radiusX: double, radiusY: double)
The x-coordinate of the center of the ellipse (default 0).
The y-coordinate of the center of the ellipse (default 0).
The horizontal radius of the ellipse (default: 0).
The vertical radius of the ellipse (default: 0).
Creates an empty Ellipse.
Creates an Ellipse with the specified center.
Creates an Ellipse with the specified center and radiuses.

FIGURE 14.33 The `Ellipse` class defines ellipses.


(centerX, centerY)
radiusX
radiusY
ShowEllipse

(a) `Ellipse(centerX, centerY, radiusX, radiusY)`

(b) Multiple ellipses are displayed.

FIGURE 14.34 An `Ellipse` object is created to display an ellipse.

LISTING 14.17 ShowEllipse.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Ellipse;
```

create a pane

main method omitted

create an ellipse

set random color for stroke

set fill color

rotate ellipse

add ellipse to pane

```
 7
 8  public class ShowEllipse extends Application {
 9    @Override // Override the start method in the Application class
10    public void start(Stage primaryStage) {
11      // Create a scene and place it in the stage
12      Scene scene = new Scene(new MyEllipse(), 300, 200);
13      primaryStage.setTitle("ShowEllipse"); // Set the stage title
14      primaryStage.setScene(scene); // Place the scene in the stage
15      primaryStage.show(); // Display the stage
16    }
17  }
18
19  class MyEllipse extends Pane {
20    private void paint() {
21      getChildren().clear();
22      for (int i = 0; i < 16; i++) {
23        // Create an ellipse and add it to pane
24        Ellipse e1 = new Ellipse(getWidth() / 2, getHeight() / 2,
25          getWidth() / 2 - 50, getHeight() / 2 - 50);
26        e1.setStroke(Color.color(Math.random(), Math.random(),
27          Math.random()));
28        e1.setFill(Color.WHITE);
29        e1.setRotate(i * 180 / 16);
30        getChildren().add(e1);
31      }
32    }
33
34    @Override
35    public void setWidth(double width) {
36      super.setWidth(width);
37      paint();
38    }
39
40    @Override
41    public void setHeight(double height) {
42      super.setHeight(height);
43      paint();
44    }
45  }
```

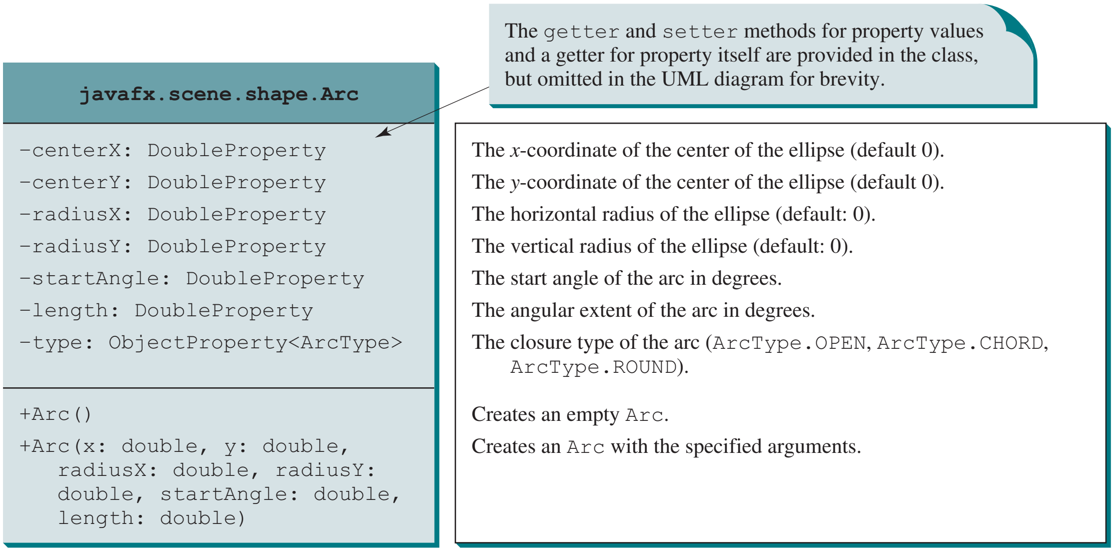
The program defines the `MyEllipse` class to draw the ellipses (lines 19–45) rather than creating ellipses directly in the `start` method (line 10) for two reasons. First, by defining the `MyEllipse` class for displaying the ellipses, you can easily reuse the code. Second, the `MyEllipse` class extends `Pane`. The contents in the pane can be resized when the stage is resized.

The `MyEllipse` class extends `Pane` and overrides the `setWidth` and `setHeight` methods (lines 34–44). A `MyEllipse` object's width and height are automatically set by invoking its `setWidth` and `setHeight` methods when it is displayed. When you resize the stage that contains a `MyEllipse`, the `MyEllipse`'s width and height are automatically resized by again invoking the `setWidth` and `setHeight` methods. The `setWidth` and `setHeight` methods invoke the `paint()` method for displaying the ellipses (lines 37 and 43). The `paint()` method first clears the contents in the pane (line 21), then repeatedly creates ellipses (lines 24 and 25), sets a random stroke color (lines 26 and 27), sets its fill color to white (line 28), rotates it (line 29), and adds the rectangle to the pane (line 30). Thus, when the stage that contains a `MyEllipse` object is resized, the contents in `MyEllipse` are redisplayed.

14.11.5 Arc

An arc is conceived as part of an ellipse, defined by the parameters **centerX**, **centerY**, **radiusX**, **radiusY**, **startAngle**, **length**, and an arc type (**ArcType.OPEN**, **ArcType.CHORD**, or **ArcType.ROUND**). The parameter **startAngle** is the starting angle, and **length** is the spanning angle (i.e., the angle covered by the arc). Angles are measured in degrees and follow the usual mathematical conventions (i.e., 0 degrees is in the easterly direction and positive angles indicate counterclockwise rotation from the easterly direction), as shown in Figure 14.36a.

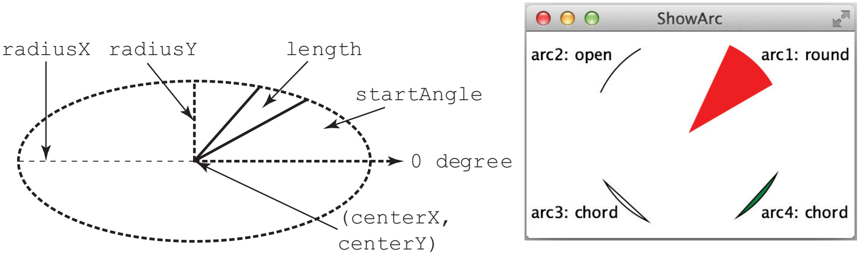
The **Arc** class defines an arc. The UML diagram for the **Arc** class is shown in Figure 14.35. Listing 14.18 gives an example that demonstrates arcs, as shown in Figure 14.36b.



The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.shape.Arc	
-centerX: DoubleProperty	The *x*-coordinate of the center of the ellipse (default 0).
-centerY: DoubleProperty	The *y*-coordinate of the center of the ellipse (default 0).
-radiusX: DoubleProperty	The horizontal radius of the ellipse (default: 0).
-radiusY: DoubleProperty	The vertical radius of the ellipse (default: 0).
-startAngle: DoubleProperty	The start angle of the arc in degrees.
-length: DoubleProperty	The angular extent of the arc in degrees.
-type: ObjectProperty<ArcType>	The closure type of the arc (`ArcType.OPEN`, `ArcType.CHORD`, `ArcType.ROUND`).
+Arc()	Creates an empty `Arc`.
+Arc(x: double, y: double, radiusX: double, radiusY: double, startAngle: double, length: double)	Creates an `Arc` with the specified arguments.

FIGURE 14.35 The **Arc** class defines an arc.



(a) `Arc(centerX, centerY, radiusX, radiusY, startAngle, length)`

(b) Multiple ellipses are displayed.

FIGURE 14.36 An **Arc** object is created to display an arc.

LISTING 14.18 ShowArc.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
```

create arc1
set fill color for arc1
set arc1 as round arc
create arc2
set fill color for arc2
set arc2 as round arc
create arc3
set fill color for arc3
set arc3 as chord arc
create arc4
create a group
add arcs and text to group
main method omitted

```
 6 import javafx.stage.Stage;
 7 import javafx.scene.shape.Arc;
 8 import javafx.scene.shape.ArcType;
 9 import javafx.scene.text.Text;
10
11 public class ShowArc extends Application {
12   @Override // Override the start method in the Application class
13   public void start(Stage primaryStage) {
14     Arc arc1 = new Arc(150, 100, 80, 80, 30, 35); // Create an arc
15     arc1.setFill(Color.RED); // Set fill color
16     arc1.setType(ArcType.ROUND); // Set arc type
17
18     Arc arc2 = new Arc(150, 100, 80, 80, 30 + 90, 35);
19     arc2.setFill(Color.WHITE);
20     arc2.setType(ArcType.OPEN);
21     arc2.setStroke(Color.BLACK);
22
23     Arc arc3 = new Arc(150, 100, 80, 80, 30 + 180, 35);
24     arc3.setFill(Color.WHITE);
25     arc3.setType(ArcType.CHORD);
26     arc3.setStroke(Color.BLACK);
27
28     Arc arc4 = new Arc(150, 100, 80, 80, 30 + 270, 35);
29     arc4.setFill(Color.GREEN);
30     arc4.setType(ArcType.CHORD);
31     arc4.setStroke(Color.BLACK);
32
33     // Create a group and add nodes to the group
34     Group group = new Group();
35     group.getChildren().addAll(new Text(210, 40, "arc1: round"),
36       arc1, new Text(20, 40, "arc2: open"), arc2,
37       new Text(20, 170, "arc3: chord"), arc3,
38       new Text(210, 170, "arc4: chord"), arc4);
39
40     // Create a scene and place it in the stage
41     Scene scene = new Scene(new BorderPane(group), 300, 200);
42     primaryStage.setTitle("ShowArc"); // Set the stage title
43     primaryStage.setScene(scene); // Place the scene in the stage
44     primaryStage.show(); // Display the stage
45   }
46 }
```

The program creates an arc `arc1` centered at (`150`, `100`) with `radiusX 80` and `radiusY 80`. The starting angle is `30` with length `35` (line 14). `arc1`'s arc type is set to `ArcType.ROUND` (line 16). Since `arc1`'s fill color is red, `arc1` is displayed filled with red round.

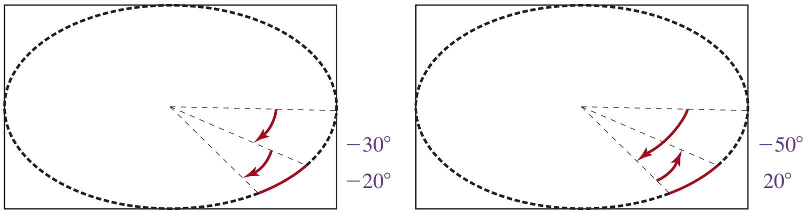
The program creates an arc `arc3` centered at (`150`, `100`) with `radiusX 80` and `radiusY 80`. The starting angle is `30+180` with length `35` (line 23). `Arc3`'s arc type is set to `ArcType.CHORD` (line 25). Since `arc3`'s fill color is white and stroke color is black, `arc3` is displayed with black outline as a chord.

negative degrees

Angles may be negative. A negative starting angle sweeps clockwise from the easterly direction, as shown in Figure 14.37. A negative spanning angle sweeps clockwise from the starting angle. The following two statements define the same arc:

```
new Arc(x, y, radiusX, radiusY, -30, -20);
new Arc(x, y, radiusX, radiusY, -50, 20);
```

The first statement uses negative starting angle `-30` and negative spanning angle `-20`, as shown in Figure 14.37a. The second statement uses negative starting angle `-50` and positive spanning angle `20`, as shown in Figure 14.37b.



(a) Negative starting angle −30° and negative spanning angle −20°

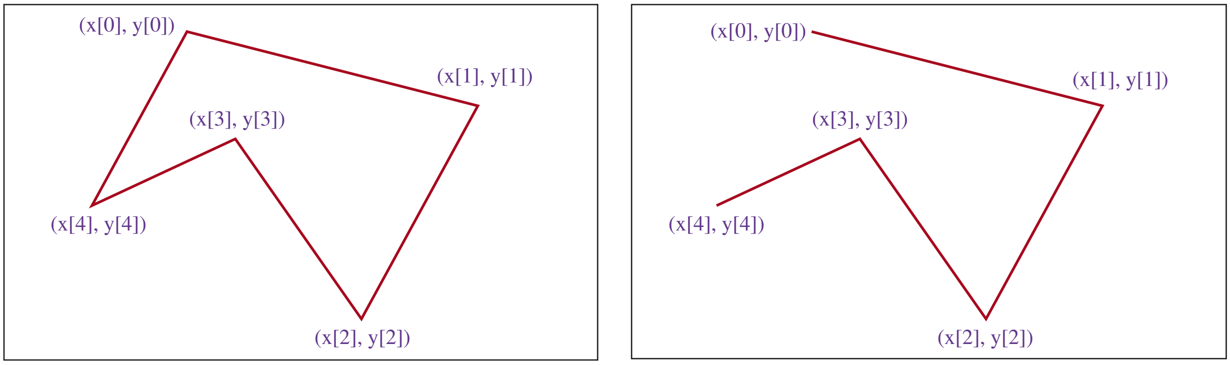
(b) Negative starting angle −50° and positive spanning angle 20°

FIGURE 14.37 Angles may be negative.

Note the trigonometric methods in the **`Math`** class use the angles in radians, but the angles in the **`Arc`** class are in degrees.

14.11.6 Polygon and Polyline

The **`Polygon`** class defines a polygon that connects a sequence of points, as shown in Figure 14.38a. The **`Polyline`** class is similar to the **`Polygon`** class except that the **`Polyline`** class is not automatically closed, as shown in Figure 14.38b.


(x[0], y[0])
(x[1], y[1])
(x[3], y[3])
(x[4], y[4])
(x[2], y[2])
(x[0], y[0])
(x[1], y[1])
(x[3], y[3])
(x[4], y[4])
(x[2], y[2])

(a) Polygon

(b) Polyline

FIGURE 14.38 **`Polygon`** is closed and **`Polyline`** is not closed.

The UML diagram for the **`Polygon`** class is shown in Figure 14.39. Listing 14.19 gives an example that creates a hexagon, as shown in Figure 14.40.

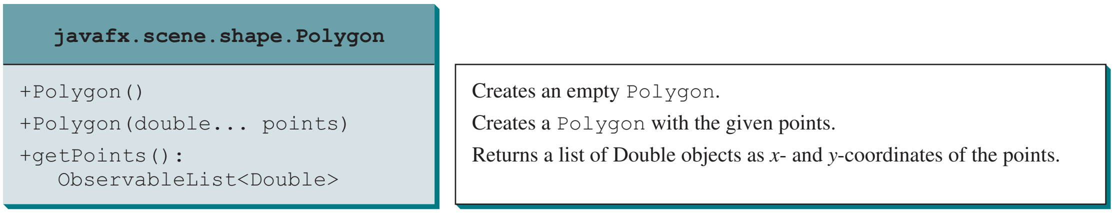
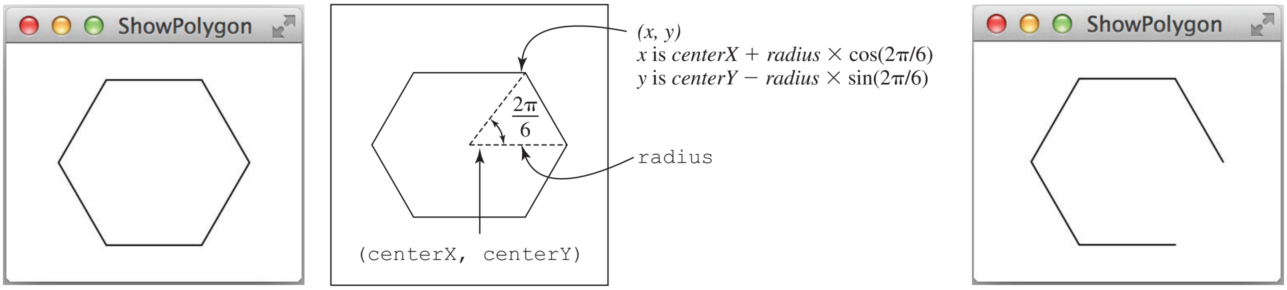

javafx.scene.shape.Polygon
+Polygon()
+Polygon(double... points)
+getPoints():
ObservableList<Double>
Creates an empty Polygon.
Creates a Polygon with the given points.
Returns a list of Double objects as x- and y-coordinates of the points.

FIGURE 14.39 The **`Polygon`** class defines a polygon.


ShowPolygon
(x, y)
x is centerX + radius × cos(2π/6)
y is centerY − radius × sin(2π/6)
2π/6
radius
(centerX, centerY)
ShowPolygon

(a)

(b)

FIGURE 14.40 (a) A `Polygon` is displayed. (b) A `Polyline` is displayed.

LISTING 14.19 ShowPolygon.java

```java
import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Polygon;

public class ShowPolygon extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place it in the stage
    Scene scene = new Scene(new MyPolygon(), 400, 400);
    primaryStage.setTitle("ShowPolygon"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}

class MyPolygon extends Pane {
  private void paint() {
    // Create a polygon and place polygon to pane
    Polygon polygon = new Polygon();
    polygon.setFill(Color.WHITE);
    polygon.setStroke(Color.BLACK);
    ObservableList<Double> list = polygon.getPoints();

    double centerX = getWidth() / 2, centerY = getHeight() / 2;
    double radius = Math.min(getWidth(), getHeight()) * 0.4;

    // Add points to the polygon list
    for (int i = 0; i < 6; i++) {
      list.add(centerX + radius * Math.cos(2 * i * Math.PI / 6));
      list.add(centerY - radius * Math.sin(2 * i * Math.PI / 6));
    }

    getChildren().clear();
    getChildren().add(polygon);
  }

  @Override
```

add pane to scene

main method omitted

extends `Pane`

create a polygon

get a list of `points`

add *x*-coordinate of a point

add *y*-coordinate of a point

```
42    public void setWidth(double width) {
43      super.setWidth(width);
44      paint();
45    }
46
47    @Override
48    public void setHeight(double height) {
49      super.setHeight(height);
50      paint();
51    }
52  }
```

override `setWidth` and `setHeight`

The program defines the `MyPolygon` class that extends `Pane` (lines 20–52). The `setWidth` and `setHeight` methods in the `Pane` class are overridden in `MyPolygon` to invoke the `paint()` method.

The `paint()` method creates a polygon (line 23) and adds it to a pane (line 38). The `polygon.getPoints()` method returns an `ObservableList<Double>` (line 26), which contains the `add` method for adding an element to the list (lines 33 and 34). Note the value passed to `add(value)` must be a `double` value. If an `int` value is passed, the `int` value would be automatically boxed into an `Integer`. This would cause an error, because the `ObservableList<Double>` consists of `Double` elements.

The `centerX`, `centerY`, and `radius` are obtained in proportion to the width and height of the pane (lines 28 and 29). The loop adds six points to the polygon (lines 32–35). Each point is represented by its *x*- and *y*-coordinates, computed using `centerX`, `centerY`, and `radius`. For each point, its *x*-coordinate is added to the polygon's list (line 33) then its *y*-coordinate is added to the list (line 34). The formula for computing the *x*- and *y*-coordinates for a point in the hexagon is illustrated in Figure 14.40a.

If you replace `Polygon` by `Polyline` (line 23), the program displays a polyline as shown in Figure 14.40b. The `Polyline` class is used in the same way as `Polygon`, except that the starting and ending points are not connected in `Polyline`.

Check Point

14.11.1 How do you display a text, line, rectangle, circle, ellipse, arc, polygon, and polyline?

14.11.2 Write code fragments to display a string rotated 45 degrees in the center of the pane.

14.11.3 Write code fragments to display a thick line of `10` pixels from (`10`, `10`) to (`70`, `30`).

14.11.4 Write code fragments to fill red color in a rectangle of width `100` and height `50` with the upper-left corner at (`10`, `10`).

14.11.5 Write code fragments to display a round-cornered rectangle with width `100`, height `200` with the upper-left corner at (`10`, `10`), corner horizontal diameter `40`, and corner vertical diameter `20`.

14.11.6 Write code fragments to display an ellipse with horizontal radius `50` and vertical radius `100`.

14.11.7 Write code fragments to display the outline of the upper half of a circle with radius `50`.

14.11.8 Write code fragments to display the lower half of a circle with radius `50` filled with the red color.

14.11.9 Write code fragments to display a polygon connecting the following points: (`20`, `40`), (`30`, `50`), (`40`, `90`), (`90`, `10`), and (`10`, `30`), and fill the polygon with green color.

14.11.10 Write code fragments to display a polyline connecting the following points: (`20`, `40`), (`30`, `50`), (`40`, `90`), (`90`, `10`), and (`10`, `30`).

14.11.11 What is wrong in the following code?

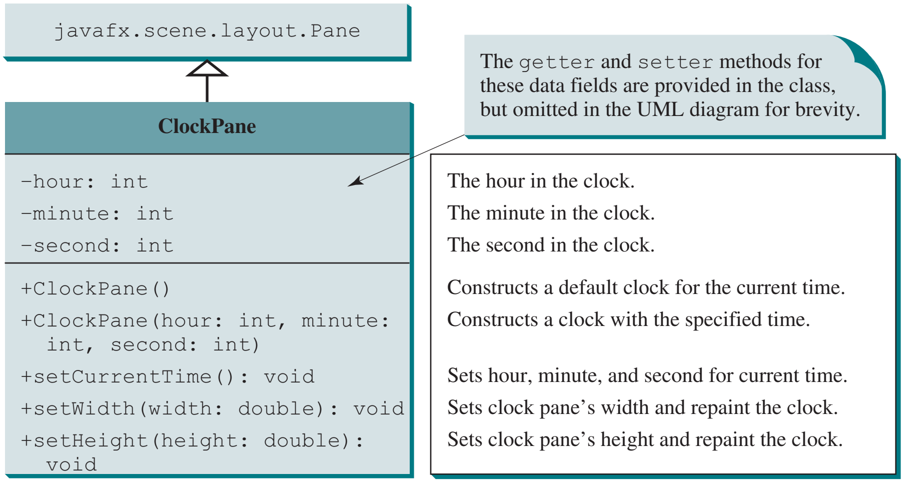
```
public void start(Stage primaryStage) {
  // Create a polygon and place it in the scene
  Scene scene = new Scene(new Polygon(), 400, 400);
  primaryStage.setScene(scene); // Place the scene in the stage
  primaryStage.show(); // Display the stage
}
```

14.12 Case Study: The **ClockPane** Class

Key Point

This case study develops a class that displays a clock on a pane.

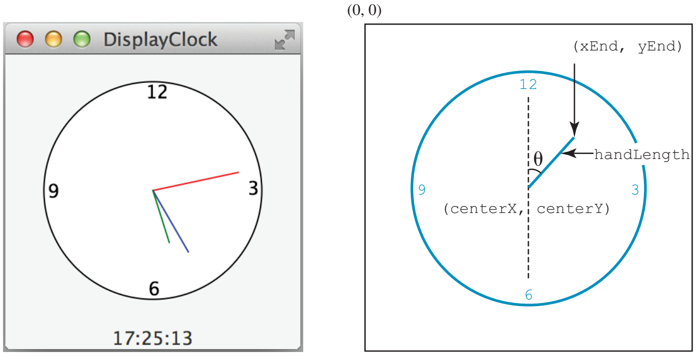
The contract of the `ClockPane` class is shown in Figure 14.41.



javafx.scene.layout.Pane
ClockPane
-hour: int
-minute: int
-second: int
+ClockPane()
+ClockPane(hour: int, minute: int, second: int)
+setCurrentTime(): void
+setWidth(width: double): void
+setHeight(height: double): void
The getter and setter methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
The hour in the clock.
The minute in the clock.
The second in the clock.
Constructs a default clock for the current time.
Constructs a clock with the specified time.
Sets hour, minute, and second for current time.
Sets clock pane's width and repaint the clock.
Sets clock pane's height and repaint the clock.

FIGURE 14.41 `ClockPane` displays an analog clock.

Assume `ClockPane` is available; we write a test program in Listing 14.20 to display an analog clock and use a label to display the hour, minute, and second, as shown in Figure 14.42a.



DisplayClock
12
9
3
6
17:25:13
(0, 0)
(xEnd, yEnd)
12
θ
handLength
9
3
(centerX, centerY)
6

(a)

(b)

FIGURE 14.42 (a) The `DisplayClock` program displays a clock that shows the current time. (b) The endpoint of a clock hand can be determined, given the spanning angle, the hand length, and the center point.

LISTING 14.20 DisplayClock.java

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;

public class DisplayClock extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a clock and a label
    ClockPane clock = new ClockPane();
    String timeString = clock.getHour() + ":" + clock.getMinute()
      + ":" + clock.getSecond();
    Label lblCurrentTime = new Label(timeString);

    // Place clock and label in border pane
    BorderPane pane = new BorderPane();
    pane.setCenter(clock);
    pane.setBottom(lblCurrentTime);
    BorderPane.setAlignment(lblCurrentTime, Pos.TOP_CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 250, 250);
    primaryStage.setTitle("DisplayClock"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

create a clock

create a label

add a clock

add a label

main method omitted

skip implementation?

implementation

The rest of this section explains how to implement the `ClockPane` class. Since you can use the class without knowing how it is implemented, you may skip the implementation if you wish.

To draw a clock, you need to draw a circle and three hands for the second, minute, and hour. To draw a hand, you need to specify the two ends of the line. As shown in Figure 14.42b, one end is the center of the clock at `(centerX, centerY)`; the other end, at `(endX, endY)`, is determined by the following formula:

```
endX = centerX + handLength × sin(θ)
endY = centerY – handLength × cos(θ)
```

Since there are 60 seconds in one minute, the angle for the second hand is

```
second × (2π/60)
```

The position of the minute hand is determined by the minute and second. The exact minute value combined with seconds is `minute + second/60`. For example, if the time is 3 minutes and 30 seconds, the total minutes are 3.5. Since there are 60 minutes in one hour, the angle for the minute hand is

$$(\text{minute} + \text{second}/60) \times (2\pi/60)$$

Since one circle is divided into 12 hours, the angle for the hour hand is

$$(\text{hour} + \text{minute}/60 + \text{second}/(60 \times 60)) \times (2\pi/12)$$

For simplicity in computing the angles of the minute and hour hands, you can omit the seconds because they are negligibly small. Therefore, the endpoints for the second, minute, and hour hands can be computed as follows:

```
secondX = centerX + secondHandLength × sin(second × (2π/60))
secondY = centerY – secondHandLength × cos(second × (2π/60))
minuteX = centerX + minuteHandLength × sin(minute × (2π/60))
minuteY = centerY – minuteHandLength × cos(minute × (2π/60))
hourX = centerX + hourHandLength × sin((hour + minute/60) × (2π/12))
hourY = centerY – hourHandLength × cos((hour + minute/60) × (2π/12))
```

The `ClockPane` class is implemented in Listing 14.21.

LISTING 14.21 ClockPane.java

```java
import java.util.Calendar;
import java.util.GregorianCalendar;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;

public class ClockPane extends Pane {
  private int hour;
  private int minute;
  private int second;

  /** Construct a default clock with the current time*/
  public ClockPane() {
    setCurrentTime();
  }

  /** Construct a clock with specified hour, minute, and second */
  public ClockPane(int hour,int minute,int second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }

  /** Return hour */
  public int getHour() {
    return hour;
  }

  /** Set a new hour */
  public void setHour(int hour) {
    this.hour = hour;
    paintClock();
  }

  /** Return minute */
  public int getMinute() {
    return minute;
  }

  /** Set a new minute */
  public void setMinute(int minute) {
    this.minute = minute;
```

clock properties

no-arg constructor

constructor

set a new hour

`paint clock`

set a new minute

```
    paintClock();
  }

  /** Return second */
  public int getSecond() {
    return second;
  }

  /** Set a new second */
  public void setSecond(int second) {
    this.second = second;
    paintClock();
  }

  /* Set the current time for the clock */
  public void setCurrentTime() {
    // Construct a calendar for the current date and time
    Calendar calendar = new GregorianCalendar();

    // Set current hour, minute and second
    this.hour = calendar.get(Calendar.HOUR_OF_DAY);
    this.minute = calendar.get(Calendar.MINUTE);
    this.second = calendar.get(Calendar.SECOND);

    paintClock(); // Repaint the clock
  }

  /** Paint the clock */
  private void paintClock() {
    // Initialize clock parameters
    double clockRadius =
      Math.min(getWidth(), getHeight()) * 0.8 * 0.5;
    double centerX = getWidth() / 2;
    double centerY = getHeight() / 2;

    // Draw circle
    Circle circle = new Circle(centerX, centerY, clockRadius);
    circle.setFill(Color.WHITE);
    circle.setStroke(Color.BLACK);
    Text t1 = new Text(centerX - 5, centerY - clockRadius + 12, "12");
    Text t2 = new Text(centerX - clockRadius + 3, centerY + 5, "9");
    Text t3 = new Text(centerX + clockRadius - 10, centerY + 3, "3");
    Text t4 = new Text(centerX - 3, centerY + clockRadius - 3, "6");

    // Draw second hand
    double sLength = clockRadius * 0.8;
    double secondX = centerX + sLength *
      Math.sin(second * (2 * Math.PI / 60));
    double secondY = centerY - sLength *
      Math.cos(second * (2 * Math.PI / 60));
    Line sLine = new Line(centerX, centerY, secondX, secondY);
    sLine.setStroke(Color.RED);

    // Draw minute hand
    double mLength = clockRadius * 0.65;
    double xMinute = centerX + mLength *
      Math.sin(minute * (2 * Math.PI / 60));
    double minuteY = centerY - mLength *
      Math.cos(minute * (2 * Math.PI / 60));
    Line mLine = new Line(centerX, centerY, xMinute, minuteY);
```

paint clock

set a new second

paint clock

set current time

paint clock

paint clock

get radius

get center

create a circle

create texts

create second hand

create minute hand

create hour hand

clear pane

add nodes to pane

set a new width

paint clock

set a new height

paint clock

```
105     mLine.setStroke(Color.BLUE);
106
107     // Draw hour hand
108     double hLength = clockRadius *  0.5;
109     double hourX = centerX + hLength *
110       Math.sin((hour % 12 + minute / 60.0) * (2 * Math.PI / 12));
111     double hourY = centerY - hLength *
112       Math.cos((hour % 12 + minute / 60.0) * (2 * Math.PI / 12));
113     Line hLine = new Line(centerX, centerY, hourX, hourY);
114     hLine.setStroke(Color.GREEN);
115
116     getChildren().clear();
117     getChildren().addAll(circle, t1, t2, t3, t4, sLine, mLine, hLine);
118   }
119
120   @Override
121   public void setWidth(double width) {
122     super.setWidth(width);
123     paintClock();
124   }
125
126   @Override
127   public void setHeight(double height) {
128     super.setHeight(height);
129     paintClock();
130   }
131 }
```

The program displays a clock for the current time using the no-arg constructor (lines 15–17) and displays a clock for the specified hour, minute, and second using the other constructor (lines 20–24).

The class defines the properties `hour`, `minute`, and `second` to store the time represented in the clock (lines 10–12). The current hour, minute, and second are obtained by using the `GregorianCalendar` class (lines 62–67). The `GregorianCalendar` class in the Java API enables you to create a `Calendar` instance for the current time using its no-arg constructor. You can then use its methods `get(Calendar.HOUR)`, `get(Calendar.MINUTE)`, and `get(Calendar.SECOND)` to return the hour, minute, and second from a `Calendar` object.

The `paintClock()` method paints the clock (lines 73–118). The clock radius is proportional to the width and height of the pane (lines 75–78). A circle for the clock is created at the center of the pane (line 81). The text for showing the hours 12, 3, 6, and 9 are created in lines 84–87. The second, minute, and hour hands are the lines created in lines 90–114. The `paintClock()` method places all these shapes in the pane using the `addAll` method (line 117). Before adding new contents into the pane, the old contents are cleared from the pane (line 116).

The `setWidth` and `setHeight` methods defined in the `Pane` class are overridden in the `ClockPane` class to repaint the clock after the width or height is changed in the clock pane (lines 120–130). The `paintClock()` method is invoked whenever a new property (`hour`, `minute`, `second`, `width`, and `height`) is set (lines 34, 45, 56, 69, 123, and 129).

override `setWidth` and `setHeight`

In Listing 14.20, the clock is placed inside a border pane, the border pane is placed in the scene, and the scene is placed in the stage. When a stage is displayed or resized, all these components inside the stage are automatically resized by invoking their respective `setWidth` and `setHeight` methods. Since the `setWidth` and `setHeight` methods are overridden to invoke the `paintClock()` method, the clock is automatically resized in response to the change of the stage size.

14.12.1 What will happen if lines 120–130 are removed in Listing 14.21? Run the DisplayClock class in Listing 14.20 to test it.

Check Point

Key Terms

AWT	542
bidirectional binding	551
bindable object	548
binding object	548
binding property	548
JavaFX	551
node	545
observable object	548
pane	545
primary stage	543
property getter method	549
shape	545
Swing	542
UI control	545
unidirectional binding	551
value getter method	549
value setter method	549

Chapter Summary

1. JavaFX is the new framework for developing rich GUI applications. JavaFX completely replaces Swing and AWT.

2. A main JavaFX class must extend `javafx.application.Application` and implement the `start` method. The primary stage is automatically created by the JVM and passed to the `start` method.

3. A stage is a window for displaying a scene. You can add nodes to a scene. Panes, groups, controls, and shapes are nodes. Panes can be used as the containers for nodes.

4. A binding property can be bound to an observable source object. A change in the source object will be automatically reflected in the binding property. A binding property has a value getter method, value setter method, and property getter method.

5. The `Node` class defines many properties that are common to all nodes. You can apply these properties to panes, groups, controls, and shapes.

6. You can create a `Color` object with the specified red, green, blue components, and opacity value.

7. You can create a `Font` object and set its name, size, weight, and posture.

8. The `javafx.scene.image.Image` class can be used to load an image, and this image can be displayed in an `ImageView` object.

9. JavaFX provides many types of panes for automatically laying out nodes in a desired location and size. The `Pane` is the base class for all panes. It contains the `getChildren()` method to return an `ObservableList`. You can use `ObservableList`'s `add(node)` and `addAll(node1, node2, ...)` methods for adding nodes into a pane.

10. A `FlowPane` arranges the nodes in the pane horizontally from left to right or vertically from top to bottom, in the order in which they were added. A `GridPane` arranges nodes

in a grid (matrix) formation. The nodes are placed in the specified column and row indices. A **`BorderPane`** can place nodes in five regions: top, bottom, left, right, and center. An **`HBox`** lays out its children in a single horizontal row. A **`VBox`** lays out its children in a single vertical column.

11. JavaFX provides many shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

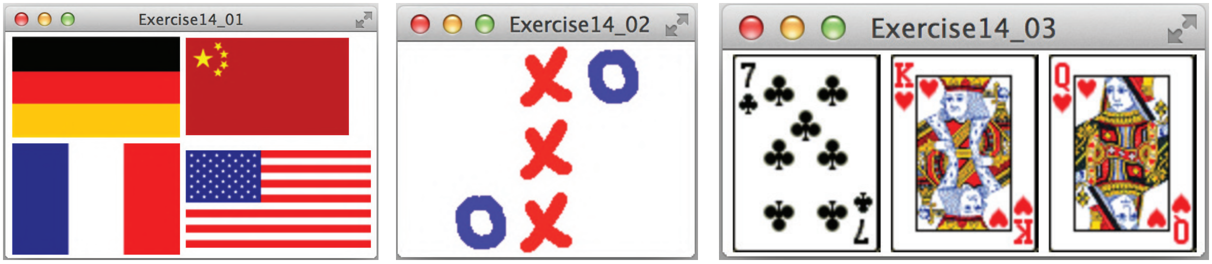
download image files

Note

The image files used in the exercises can be obtained from liveexample.pearsoncmg.com/resource/image.zip under the image folder.

Sections 14.2–14.9

14.1 (*Display images*) Write a program that displays four images in a grid pane, as shown in Figure 14.43a.



(a) (b) (c)

FIGURE 14.43 (a) Exercise 14.1 displays four images. *Source*: booka/Fotolia. Figure 14.43a4: United States Government. (b) Exercise 14.2 displays a tic-tac-toe board with images. (c) Three cards are randomly selected. *Source*: pandawild/Fotolia.

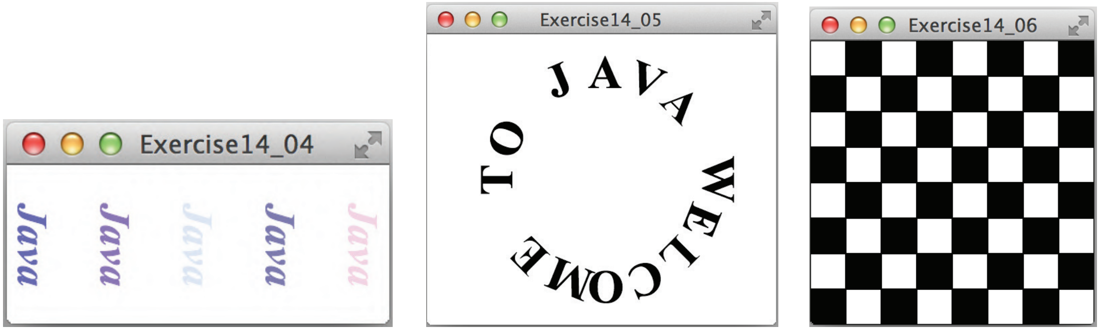
VideoNote

Display a tic-tac-toe board

***14.2** (*Tic-tac-toe board*) Write a program that displays a tic-tac-toe board, as shown in Figure 14.43b. A cell may be X, O, or empty. What to display at each cell is randomly decided. The X and O are the image files **x.gif** and **o.gif**.

***14.3** (*Display three cards*) Write a program that displays three cards randomly selected from a deck of 52, as shown in Figure 14.43c. The card image files are named **1.png**, **2.png**, . . . , **52.png** and stored in the **image/card** directory. All three cards are distinct and selected randomly. (*Hint*: You can select random cards by storing the numbers 1–52 to an array list, perform a random shuffle introduced in Section 11.12, and use the first three numbers in the array list as the file names for the image.)

14.4 (*Color and font*) Write a program that displays five texts vertically, as shown in Figure 14.44a. Set a random color and opacity for each text and set the font of each text to Times Roman, bold, italic, and 22 pixels.



(a) (b) (c)

FIGURE 14.44 (a) Five texts are displayed with a random color and a specified font. (b) A string is displayed around the circle. (c) A checkerboard is displayed using rectangles.

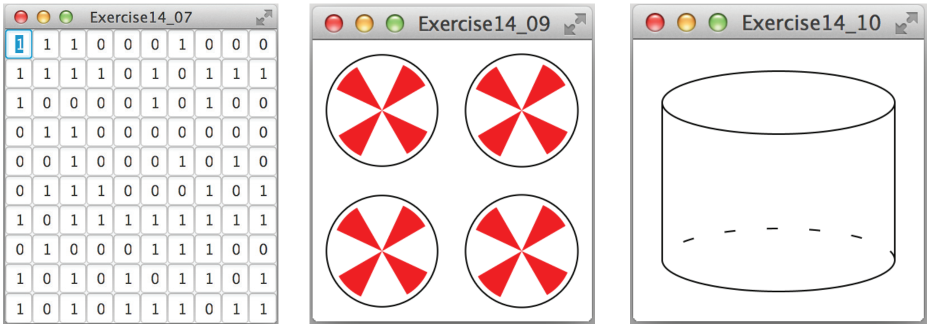
14.5 (*Characters around circle*) Write a program that displays a string "Welcome to Java" around the circle, as shown in Figure 14.44b. (*Hint*: You need to display each character in the right location with appropriate rotation using a loop.)

***14.6** (*Game: display a checkerboard*) Write a program that displays a checkerboard in which each white and black cell is a `Rectangle` with a fill color black or white, as shown in Figure 14.44c.

Sections 14.10 and 14.11

***14.7** (*Display random 0 or 1*) Write a program that displays a 10-by-10 square matrix, as shown in Figure 14.45a. Each element in the matrix is `0` or `1`, randomly generated. Display each number centered in a text field. Use `TextField`'s `setText` method to set value `0` or `1` as a string.

Display a random matrix



1	1	1	0	0	0	1	0	0	0
1	1	1	1	0	1	0	1	1	1
1	0	0	0	0	1	0	1	0	0
0	1	1	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	1	0
0	1	1	1	0	0	0	1	0	1
1	0	1	1	1	1	1	1	1	1
0	1	0	0	0	1	1	1	0	0
1	0	1	0	1	0	1	1	0	1
1	0	1	0	1	1	1	0	1	1

(a) (b) (c)

FIGURE 14.45 (a) The program randomly generates 0s and 1s. (b) Exercise 14.9 draws four fans. (c) Exercise 14.10 draws a cylinder.

14.8 (*Display 54 cards*) Expand Exercise 14.3 to display all 54 cards (including two jokers), nine per row. The image files are jokers and are named 53.png and 54.png.

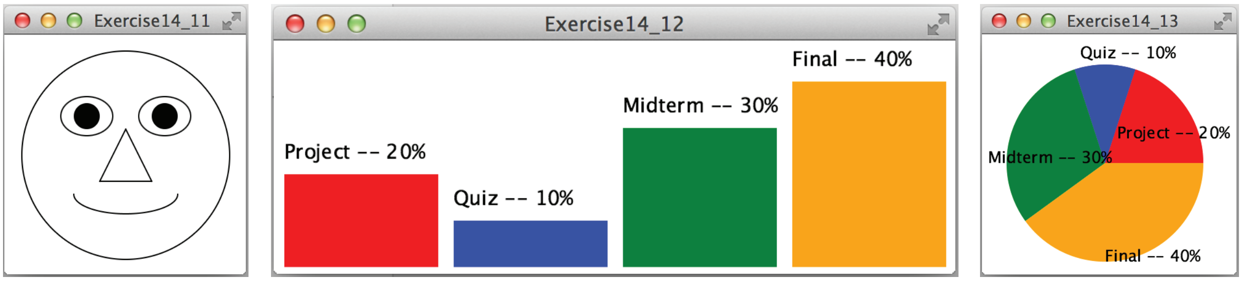
***14.9** (*Create four fans*) Write a program that places four fans in a `GridPane` with two rows and two columns, as shown in Figure 14.45b.

***14.10** (*Display a cylinder*) Write a program that draws a cylinder, as shown in Figure 14.45c. You can use the following method to set the dashed stroke for an arc:

```
arc.getStrokeDashArray().addAll(6.0, 21.0);
```

The solution posted on the website enables the cylinder to resize horizontally. Can you revise it to resize vertically as well?

***14.11** (*Paint a smiley face*) Write a program that paints a smiley face, as shown in Figure 14.46a.



(a) (b) (c)

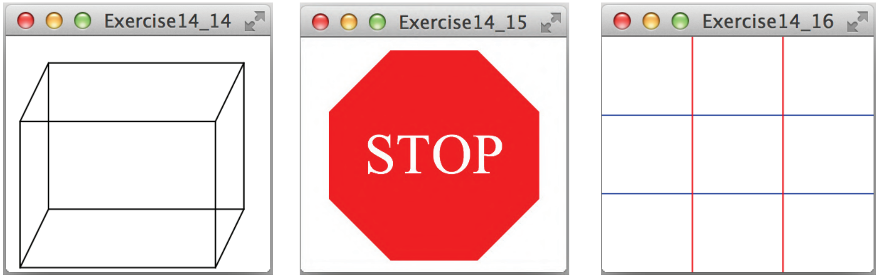
Figure 14.46 (a) Exercise 14.11 paints a smiley face. (b) Exercise 14.12 paints a bar chart. (c) Exercise 14.13 paints a pie chart.

VideoNote
Display a bar chart

****14.12** (*Display a bar chart*) Write a program that uses a bar chart to display the percentages of the overall grade represented by projects, quizzes, midterm exams, and the final exam, as shown in Figure 14.46b. Suppose projects take `20`% and are displayed in red, quizzes take `10`% and are displayed in blue, midterm exams take `30`% and are displayed in green, and the final exam takes `40`% and is displayed in orange. Use the `Rectangle` class to display the bars. Interested readers may explore the JavaFX `BarChart` class for further study.

****14.13** (*Display a pie chart*) Write a program that uses a pie chart to display the percentages of the overall grade represented by projects, quizzes, midterm exams, and the final exam, as shown in Figure 14.46c. Suppose projects take `20`% and are displayed in red, quizzes take `10`% and are displayed in blue, midterm exams take `30`% and are displayed in green, and the final exam takes `40`% and is displayed in orange. Use the `Arc` class to display the pies. Interested readers may explore the JavaFX `PieChart` class for further study.

14.14 (*Display a rectanguloid*) Write a program that displays a rectanguloid, as shown in Figure 14.47a. The cube should grow and shrink as the window grows or shrinks.



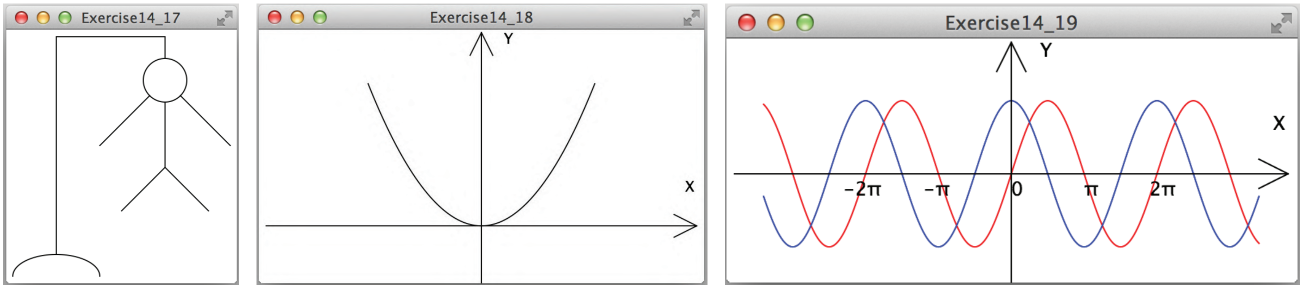
(a) (b) (c)

Figure 14.47 (a) Exercise 14.14 paints a rectanguloid. (b) Exercise 14.15 paints a STOP sign. (c) Exercise 14.16 paints a grid.

***14.15** (*Display a STOP sign*) Write a program that displays a STOP sign, as shown in Figure 14.47b. The octagon is in red and the sign is in white. (*Hint*: Place an octagon and a text in a stack pane.)

***14.16** (*Display a* 3×3 *grid*) Write a program that displays a 3×3 grid, as shown in Figure 14.47c. Use red color for vertical lines and blue for horizontals. The lines are automatically resized when the window is resized.

14.17 (*Game: hangman*) Write a program that displays a drawing for the popular hangman game, as shown in Figure 14.48a.



(a) (b) (c)

FIGURE 14.48 (a) Exercise 14.17 draws a sketch for the hangman game. (b) Exercise 14.18 plots the quadratic function. (c) Exercise 14.19 plots the sine/cosine functions.

***14.18** (*Plot the square function*) Write a program that draws a diagram for the function $f(x) = x^2$ (see Figure 14.48b).

Hint: Add points to a polyline using the following code:

```
Polyline polyline = new Polyline();
ObservableList<Double> list = polyline.getPoints();
double scaleFactor = 0.0125;
for (int x = -100; x <= 100; x++) {
  list.add(x + 200.0);
  list.add(scaleFactor * x * x);
}
```

****14.19** (*Plot the sine and cosine functions*) Write a program that plots the sine function in red and cosine in blue, as shown in Figure 14.48c.

Hint: The Unicode for π is `\u03c0`. To display -2π, use `Text(x, y, "-2\u03c0")`. For a trigonometric function like `sin(x)`, `x` is in radians. Use the following loop to add the points to a polyline:

```
Polyline polyline = new Polyline();
ObservableList<Double> list = polyline.getPoints();
double scaleFactor = 50;
for (int x = -170; x <= 170; x++) {
  list.add(x + 200.0);
  list.add(100 - scaleFactor * Math.sin((x / 100.0) * 2 *
  Math.PI));
}
```

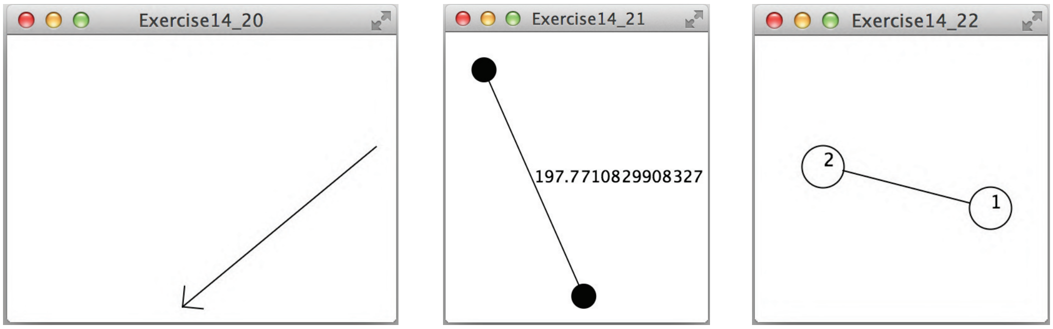
Note that `x` in the loop is a point in the X-Axis and `x` does not correspond to angles in degrees. The entire expression `(x / 100.0) * 2 * Math.PI` represents an angle in radians.

```
When x is -100, Math.sin((x / 100.0) * 2 * Math.PI) is 0
When x is -75, Math.sin((x / 100.0) * 2 * Math.PI) is 1
When x is -50, Math.sin((x / 100.0) * 2 * Math.PI) is 0
When x is -25, Math.sin((x / 100.0) * 2 * Math.PI) is -1
When x is 0, Math.sin((x / 100.0) * 2 * Math.PI) is 0
When x is 25, Math.sin((x / 100.0) * 2 * Math.PI) is 1
When x is 50, Math.sin((x / 100.0) * 2 * Math.PI) is 0
When x is 75, Math.sin((x / 100.0) * 2 * Math.PI) is -1
When x is 100, Math.sin((x / 100.0) * 2 * Math.PI) is 0
```

****14.20** (*Draw an arrow line*) Write a static method that draws an arrow line from a starting point to an ending point in a pane using the following method header:

```
public static void drawArrowLine(double startX, double startY,
  double endX, double endY, Pane pane)
```

Write a test program that randomly draws an arrow line, as shown in Figure 14.49a.



(a) (b) (c)

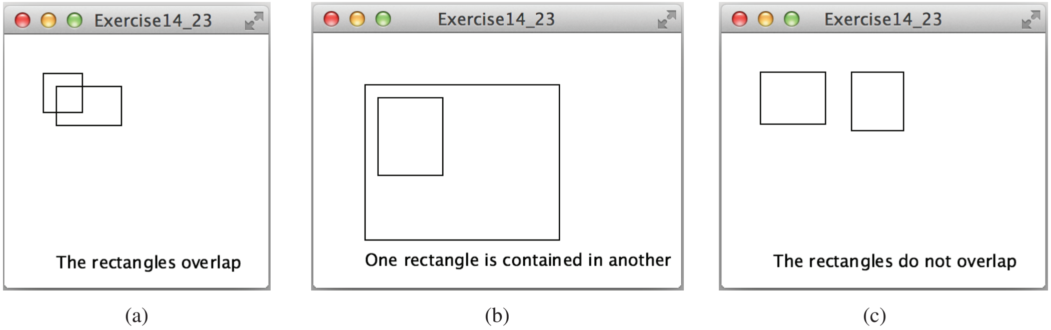
FIGURE 14.49 (a) The program displays an arrow line. (b) Exercise 14.21 connects the centers of two filled circles. (c) Exercise 14.22 connects two circles from their perimeter.

***14.21** (*Two circles and their distance*) Write a program that draws two circles with radius **15** pixels, centered at random locations, with a line connecting the two circles. The distance between the two centers is displayed on the line, as shown in Figure 14.49b.

***14.22** (*Connect two circles*) Write a program that draws two filled circles with radius **15** pixels, centered at random locations, with a line connecting the two circles. The line should not cross inside the circles, as shown in Figure 14.49c.

***14.23** (*Geometry: two rectangles*) Write a program that prompts the user to enter the center coordinates, width, and height of two rectangles from the command line. The program displays the rectangles and a text indicating whether the two are overlapping, whether one is contained in the other, or whether they don't overlap, as shown in Figure 14.50. See Programming Exercise 10.13 for checking the relationship between two rectangles.

***14.24** (*Geometry: Inside a polygon?*) Write a program that prompts the user to enter the coordinates of five points from the command line. The first four points form a polygon, and the program displays the polygon and a text that indicates whether the fifth point is inside the polygon, as shown in Figure 14.51a. (*Hint*: Use the `Node`'s `contains` method to test whether a point is inside a node.)



The rectangles overlap

One rectangle is contained in another

The rectangles do not overlap

(a) (b) (c)

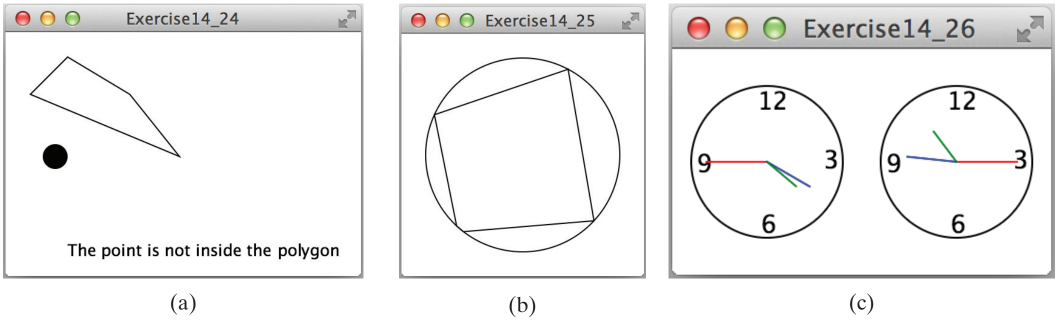
FIGURE 14.50 Two rectangles are displayed.

***14.25** (*Random points on a circle*) Modify Programming Exercise 4.6 to create five random points on a circle, form a polygon by connecting the points clockwise, and display the circle and the polygon, as shown in Figure 14.51b.

Section 14.12

14.26 (*Use the `ClockPane` class*) Write a program that displays two clocks. The hour, minute, and second values are `4`, `20`, `45` for the first clock, and `22`, `46`, `15` for the second clock, as shown in Figure 14.51c.

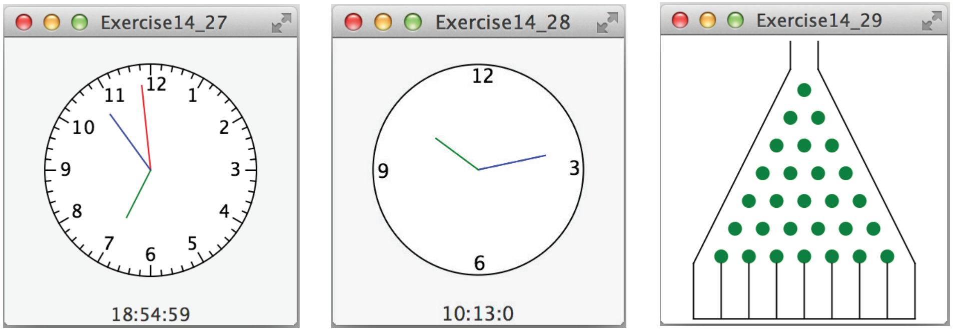
***14.27** (*Draw a detailed clock*) Modify the `ClockPane` class in Section 14.12 to draw the clock with more details on the hours and minutes, as shown in Figure 14.52a.



The point is not inside the polygon

(a) (b) (c)

FIGURE 14.51 (a) The polygon and a point are displayed. (b) Exercise 14.25 connects five random points on a circle. (c) Exercise 14.26 displays two clocks.



(a) (b) (c)

FIGURE 14.52 (a) Exercise 14.27 displays a detailed clock. (b) Exercise 14.28 displays a clock with random hour and minute values. (c) Exercise 14.29 displays a bean machine.

***14.28** (*Random time*) Modify the `ClockPane` class with three new Boolean properties—`hourHandVisible`, `minuteHandVisible`, and `secondHandVisible`—and their associated accessor and mutator methods. You can use the `set` methods to make a hand visible or invisible. Write a test program that displays only the hour and minute hands. The hour and minute values are randomly generated. The hour is between `0` and `11`, and the minute is either `0` or `30`, as shown in Figure 14.52b.

****14.29** (*Game: bean machine*) Write a program that displays a bean machine introduced in Programming Exercise 7.37, as shown in Figure 14.52c.

CHAPTER

15

EVENT-DRIVEN PROGRAMMING AND ANIMATIONS

Objectives

- To get a taste of event-driven programming (§15.1).
- To describe events, event sources, and event classes (§15.2).
- To define handler classes, register handler objects with the source object, and write the code to handle events (§15.3).
- To define handler classes using inner classes (§15.4).
- To define handler classes using anonymous inner classes (§15.5).
- To simplify event handling using lambda expressions (§15.6).
- To develop a GUI application for a loan calculator (§15.7).
- To write programs to deal with `MouseEvent`s (§15.8).
- To write programs to deal with `KeyEvent`s (§15.9).
- To create listeners for processing a value change in an observable object (§15.10).
- To use the `Animation`, `PathTransition`, `FadeTransition`, and `Timeline` classes to develop animations (§15.11).
- To develop an animation for simulating a bouncing ball (§15.12).
- To draw, color, and resize a US map (§15.13).

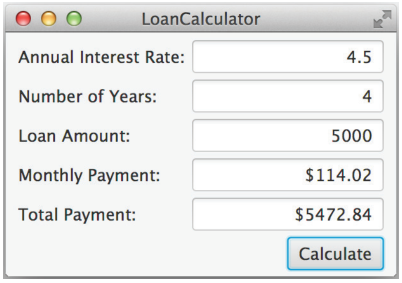
15.1 Introduction

Key Point

You can write code to process events such as a button click, mouse movement, and keystrokes.

problem

Suppose you wish to write a GUI program that lets the user enter a loan amount, annual interest rate, and number of years then click the *Calculate* button to obtain the monthly payment and total payment, as shown in Figure 15.1. How do you accomplish the task? You have to use *event-driven programming* to write the code to respond to the button-clicking event.



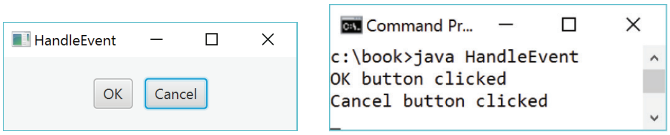
LoanCalculator

Annual Interest Rate: 4.5
Number of Years: 4
Loan Amount: 5000
Monthly Payment: $114.02
Total Payment: $5472.84

Calculate

FIGURE 15.1 The program computes loan payments.

Before delving into event-driven programming, it is helpful to get a taste using a simple example. The example displays two buttons in a pane, as shown in Figure 15.2.



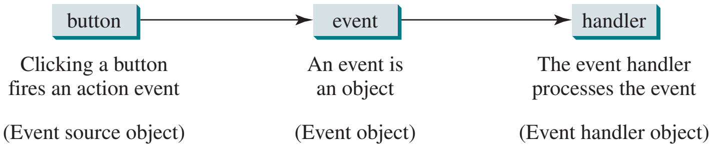
HandleEvent
OK
Cancel
Command Pr...

```
c:\book>java HandleEvent
OK button clicked
Cancel button clicked
```

(a) (b)

FIGURE 15.2 (a) The program displays two buttons. (b) A message is displayed in the console when a button is clicked.

To respond to a button click, you need to write the code to process the button-clicking action. The button is an *event source object*—where the action originates. You need to create an object capable of handling the action event on a button. This object is called an *event handler*, as shown in Figure 15.3.



button
event
handler
Clicking a button fires an action event
An event is an object
The event handler processes the event
(Event source object)
(Event object)
(Event handler object)

FIGURE 15.3 An event handler processes the event fired from the source object.

Not all objects can be handlers for an action event. To be a handler of an action event, two requirements must be met:

`EventHandler` interface

1. The object must be an instance of the `EventHandler<T extends Event>` interface. This interface defines the common behavior for all handlers. `<T extends Event>` denotes that `T` is a generic type that is a subtype of `Event`.

`setOnAction(handler)`

2. The `EventHandler` object `handler` must be registered with the event source object using the method `source.setOnAction(handler)`.

The **`EventHandler<ActionEvent>`** interface contains the **`handle(ActionEvent)`** method for processing the action event. Your handler class must override this method to respond to the event. Listing 15.1 gives the code that processes the **`ActionEvent`** on the two buttons. When you click the *OK* button, the message "OK button clicked" is displayed. When you click the *Cancel* button, the message "Cancel button clicked" is displayed, as shown in Figure 15.2.

LISTING 15.1 HandleEvent.java

```
 1  import javafx.application.Application;
 2  import javafx.geometry.Pos;
 3  import javafx.scene.Scene;
 4  import javafx.scene.control.Button;
 5  import javafx.scene.layout.HBox;
 6  import javafx.stage.Stage;
 7  import javafx.event.ActionEvent;
 8  import javafx.event.EventHandler;
 9
10  public class HandleEvent extends Application {
11    @Override // Override the start method in the Application class
12    public void start(Stage primaryStage) {
13      // Create a pane and set its properties
14      HBox pane = new HBox(10);
15      pane.setAlignment(Pos.CENTER);
16      Button btOK = new Button("OK");
17      Button btCancel = new Button("Cancel");
18      OKHandlerClass handler1 = new OKHandlerClass();
19      btOK.setOnAction(handler1);
20      CancelHandlerClass handler2 = new CancelHandlerClass();
21      btCancel.setOnAction(handler2);
22      pane.getChildren().addAll(btOK, btCancel);
23
24      // Create a scene and place it in the stage
25      Scene scene = new Scene(pane);
26      primaryStage.setTitle("HandleEvent"); // Set the stage title
27      primaryStage.setScene(scene); // Place the scene in the stage
28      primaryStage.show(); // Display the stage
29    }
30  }
31
32  class OKHandlerClass implements EventHandler<ActionEvent> {
33    @Override
34    public void handle(ActionEvent e) {
35      System.out.println("OK button clicked");
36    }
37  }
38
39  class CancelHandlerClass implements EventHandler<ActionEvent> {
40    @Override
41    public void handle(ActionEvent e) {
42      System.out.println("Cancel button clicked");
43    }
44  }
```

create handler
register handler
create handler
register handler
main method omitted
handler class
handle event
handler class
handle event

Two handler classes are defined in lines 32-44. Each handler class implements **`EventHandler<ActionEvent>`** to process **`ActionEvent`**. The object **`handler1`** is an instance of **`OKHandlerClass`** (line 18), which is registered with the button **`btOK`** (line 19). When the *OK* button is clicked, the **`handle(ActionEvent)`** method (line 34) in **`OKHandlerClass`** is invoked to process the event. The object **`handler2`** is an instance of **`CancelHandlerClass`** (line 20), which is registered with the button **`btCancel`** in line 21. When the *Cancel* button is clicked, the **`handle(ActionEvent)`** method (line 41) in **`CancelHandlerClass`** is invoked to process the event.

You now have seen a glimpse of event-driven programming in JavaFX. You probably have many questions, such as why a handler class is defined to implement the **`EventHandler<ActionEvent>`**. The following sections will give you all the answers.

15.2 Events and Event Sources

Key Point

An event is an object created from an event source. Firing an event means to create an event and delegate the handler to handle the event.

event-driven programming
event

When you run a Java GUI program, the program interacts with the user and the events drive its execution. This is called *event-driven programming*. An *event* can be defined as a signal to the program that something has happened. Events are triggered by external user actions, such as mouse movements, mouse clicks, and keystrokes. The program can choose to respond to or ignore an event. The example in the preceding section gave you a taste of event-driven programming.

fire event
event source object
source object

The component that creates an event and fires it is called the *event source object*, or simply *source object* or *source component*. For example, a button is the source object for a button-clicking action event. An event is an instance of an event class. The root class of the Java event classes is **`java.util.EventObject`**. The root class of the JavaFX event classes is **`javafx.event.Event`**. The hierarchical relationships of some event classes are shown in Figure 15.4.

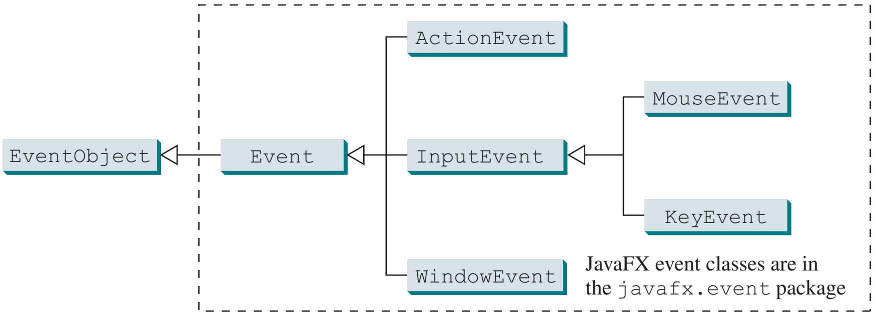


FIGURE 15.4 An event in JavaFX is an object of the **`javafx.event.Event`** class.

event object
`getSource()`

An *event object* contains whatever properties are pertinent to the event. You can identify the source object of an event using the **`getSource()`** instance method in the **`EventObject`** class. The subclasses of **`EventObject`** deal with specific types of events, such as action events, window events, mouse events, and key events. The first three columns in Table 15.1 list some external user actions, source objects, and event types fired. For example, when clicking a button, the button creates and fires an **`ActionEvent`**, as indicated in the first line of this table. Here, the button is an event source object, and an **`ActionEvent`** is the event object fired by the source object, as shown in Figure 15.3.

Note

If a component can fire an event, any subclass of the component can fire the same type of event. For example, every JavaFX shape, layout pane, and control can fire **`MouseEvent`** and **`KeyEvent`** since **`Node`** is the superclass for shapes, layout panes, and controls and **`Node`** can fire **`MouseEvent`** and **`KeyEvent`**.

TABLE 15.1 User Action, Source Object, Event Type, Handler Interface, and Handler

User Action	*Source Object*	*Event Type Fired*	*Event Registration Method*
Click a button	`Button`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Press Enter in a text field	`TextField`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`RadioButton`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`CheckBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Select a new item	`ComboBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Mouse pressed	`Node, Scene`	`MouseEvent`	`setOnMousePressed(EventHandler<MouseEvent>)`
Mouse released			`setOnMouseReleased(EventHandler<MouseEvent>)`
Mouse clicked			`setOnMouseClicked(EventHandler<MouseEvent>)`
Mouse entered			`setOnMouseEntered(EventHandler<MouseEvent>)`
Mouse exited			`setOnMouseExited(EventHandler<MouseEvent>)`
Mouse moved			`setOnMouseMoved(EventHandler<MouseEvent>)`
Mouse dragged			`setOnMouseDragged(EventHandler<MouseEvent>)`
Key pressed	`Node, Scene`	`KeyEvent`	`setOnKeyPressed(EventHandler<KeyEvent>)`
Key released			`setOnKeyReleased(EventHandler<KeyEvent>)`
Key typed			`setOnKeyTyped(EventHandler<KeyEvent>)`

Check Point

15.2.1 What is an event source object? What is an event object? Describe the relationship between an event source object and an event object.

15.2.2 Can a button fire a `MouseEvent`? Can a button fire a `KeyEvent`? Can a button fire an `ActionEvent`?

15.3 Registering Handlers and Handling Events

Key Point

A handler is an object that must be registered with an event source object and it must be an instance of an appropriate event-handling interface.

event delegation
event handler

Java uses a delegation-based model for event handling: A source object fires an event, and an object interested in the event handles it. The latter object is called an *event handler* or an event *listener*. For an object to be a handler for an event on a source object, two things are needed, as shown in Figure 15.5.

event–handler interface

`EventHandler<T extends Event>`

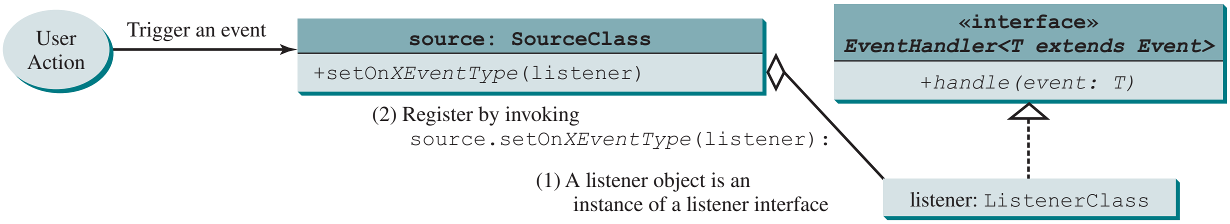
event handler

1. *The handler object must be an instance of the corresponding event–handler interface* to ensure the handler has the correct method for processing the event. JavaFX defines a unified handler interface `EventHandler<T extends Event>` for an event `T`. The handler interface contains the `handle(T e)` method for processing the event. For example, the handler interface for `ActionEvent` is `EventHandler<ActionEvent>`; each handler for `ActionEvent` should implement the `handle(ActionEvent e)` method for processing an `ActionEvent`.

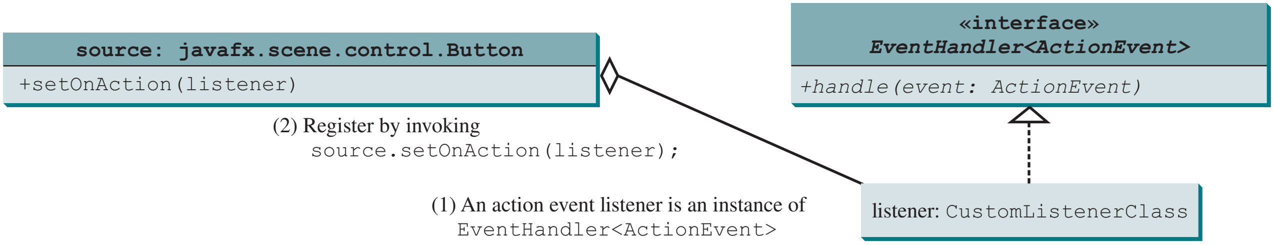
register handler

2. *The handler object must be registered by the source object*. Registration methods depend on the event type. For `ActionEvent`, the method is `setOnAction`. For a mouse-pressed event, the method is `setOnMousePressed`. For a key-pressed event, the method is `setOnKeyPressed`.

Let's revisit Listing 15.1, HandleEvent.java. Since a `Button` object fires `ActionEvent`, a handler object for `ActionEvent` must be an instance of `EventHandler<ActionEvent>`, so


User Action
Trigger an event
source: SourceClass
+setOnXEventType(listener)
«interface»
EventHandler<T extends Event>
+handle(event: T)
(2) Register by invoking
source.setOnXEventType(listener):
(1) A listener object is an
instance of a listener interface
listener: ListenerClass

(a) A generic source object with a generic event `T`


source: javafx.scene.control.Button
+setOnAction(listener)
«interface»
EventHandler<ActionEvent>
+handle(event: ActionEvent)
(2) Register by invoking
source.setOnAction(listener);
(1) An action event listener is an instance of
EventHandler<ActionEvent>
listener: CustomListenerClass

(b) A `Button` source object with an `ActionEvent`

FIGURE 15.5 A listener must be an instance of a listener interface and must be registered with a source object.

the handler class implements **`EventHandler<ActionEvent>`** in line 32. The source object invokes **`setOnAction(handler)`** to register a handler, as follows:

create source object

```
// Line 16 in Listing 15.1
Button btOK = new Button("OK");
```

create handler object

```
// Line 18 in Listing 15.1
OKHandlerClass handler1 = new OKHandlerClass();
```

register handler

```
// Line 19 in Listing 15.1
btOK.setOnAction(handler1);
```

When you click the button, the **`Button`** object fires an **`ActionEvent`** and passes it to invoke the handler's **`handle(ActionEvent)`** method to handle the event. The event object contains information pertinent to the event, which can be obtained using the methods. For example, you can use **`e.getSource()`** to obtain the source object that fired the event.

first version

We now write a program that uses two buttons to enlarge and shrink a circle, as shown in Figure 15.6. We will develop this program incrementally. First, we write the program in Listing 15.2 that displays the user interface with a circle in the center (lines 15–19) and two buttons on the bottom (lines 21–27).

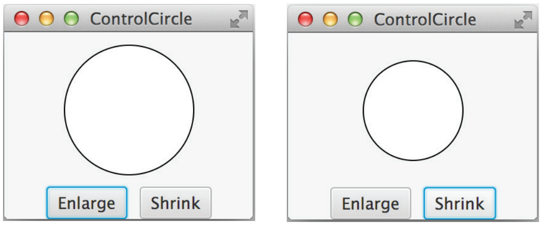

ControlCircle
Enlarge
Shrink
ControlCircle
Enlarge
Shrink

FIGURE 15.6 The user clicks the *Enlarge* and *Shrink* buttons to enlarge and shrink the circle.

LISTING 15.2 ControlCircleWithoutEventHandling.java

```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.scene.Scene;
```

```java
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ControlCircleWithoutEventHandling extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    StackPane pane = new StackPane();
    Circle circle = new Circle(50);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);
    pane.getChildren().add(circle);

    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(pane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 200, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

circle

buttons

main method omitted

second version

How do you use the buttons to enlarge or shrink the circle? When the *Enlarge* button is clicked, you want the circle to be repainted with a larger radius. How can you accomplish this? You can expand and modify the program in Listing 15.2 into Listing 15.3 with the following features:

1. Define a new class named `CirclePane` for displaying the circle in a pane (lines 51–68). This new class displays a circle and provides the `enlarge` and `shrink` methods for increasing and decreasing the radius of the circle (lines 60–62 and 64–67). It is a good strategy to design a class to model a circle pane with supporting methods so these related methods along with the circle are coupled in one object.

2. Create a `CirclePane` object and declare `circlePane` as a data field to reference this object (line 15) in the `ControlCircle` class. The methods in the `ControlCircle` class can now access the `CirclePane` object through this data field.

3. Define a handler class named `EnlargeHandler` that implements `EventHandler<ActionEvent>` (lines 43–48). To make the reference variable `circlePane` accessible from the `handle` method, define `EnlargeHandler` as an inner class of the `ControlCircle` class. (*Inner classes* are defined inside another class. We use an inner class here and will introduce it fully in the next section.)

inner class

4. Register the handler for the *Enlarge* button (line 29) and implement the `handle` method in `EnlargeHandler` to invoke `circlePane.enlarge()` (line 46).

VideoNote
Handler and its registration

LISTING 15.3 ControlCircle.java

```java
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ControlCircle extends Application {
  private CirclePane circlePane = new CirclePane();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Hold two buttons in an HBox
    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    // Create and register the handler
    btEnlarge.setOnAction(new EnlargeHandler());

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(circlePane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 200, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  class EnlargeHandler implements EventHandler<ActionEvent> {
    @Override // Override the handle method
    public void handle(ActionEvent e) {
      circlePane.enlarge();
    }
  }
}

class CirclePane extends StackPane {
  private Circle circle = new Circle(50);

  public CirclePane() {
    getChildren().add(circle);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);
  }
```

create/register handler

handler class

main method omitted

`CirclePane` class

```

  public void enlarge() {
    circle.setRadius(circle.getRadius() + 2);
  }

  public void shrink() {
    circle.setRadius(circle.getRadius() > 2 ?
      circle.getRadius() - 2 : circle.getRadius());
  }
}
```

enlarge method

As an exercise, add the code for handling the *Shrink* button to display a smaller circle when the *Shrink* button is clicked.

the *Shrink* button

Check Point

15.3.1 Why must a handler be an instance of an appropriate handler interface?

15.3.2 Explain how to register a handler object and how to implement a handler interface.

15.3.3 What is the handler method for the `EventHandler<ActionEvent>` interface?

15.3.4 What is the registration method for a button to register an `ActionEvent` handler?

15.4 Inner Classes

Key Point

An inner class, or nested class, is a class defined within the scope of another class. Inner classes are useful for defining handler classes.

The approach of this book is to introduce difficult programming concepts using practical examples. We introduce inner classes, anonymous inner classes, and lambda expressions using practical examples in this section and following two sections.

Inner classes are used in the preceding section. This section introduces inner classes in detail. First, let us see the code in Figure 15.7. The code in Figure 15.7a defines two separate classes, `Test` and `A`. The code in Figure 15.7b defines `A` as an inner class in `Test`.

```
public class Test {
  ...
}

public class A {
  ...
}
```

(a)

```
public class Test {
  ...

  // Inner class
  public class A {
    ...
  }
}
```

(b)

```
// OuterClass.java: inner class demo
public class OuterClass {
  private int data;

  /** A method in the outer class */
  public void m() {
    // Do something
  }

  // An inner class
  class InnerClass {
    /** A method in the inner class */
    public void mi() {
      // Directly reference data and method
      // defined in its outer class
      data++;
      m();
    }
  }
}
```

(c)

Figure 15.7 An inner class is defined as a member of another class.

The class `InnerClass` defined inside `OuterClass` in Figure 15.7c is another example of an inner class. An inner class may be used just like a regular class. Normally, you define a class as an inner class if it is used only by its outer class. An inner class has the following features:

- An inner class is compiled into a class named `OuterClassName$InnerClassName.class`. For example, the inner class `A` in `Test` is compiled into `Test$A.class` in Figure 15.7b.
- An inner class can reference the data and the methods defined in the outer class in which it nests, so you need not pass the reference of an object of the outer class to the constructor of the inner class. For this reason, inner classes can make programs simple and concise. For example, `circlePane` is defined in `ControlCircle` in Listing 15.3 (line 15). It can be referenced in the inner class `EnlargeHandler` in line 46.
- An inner class can be defined with a visibility modifier subject to the same visibility rules applied to a member of the class.
- An inner class can be defined as `static`. A `static` inner class can be accessed using the outer class name. A `static` inner class cannot access nonstatic members of the outer class.
- Objects of an inner class are often created in the outer class. However, you can also create an object of an inner class from another class. If the inner class is nonstatic, you must first create an instance of the outer class, then use the following syntax to create an object for the inner class:

  ```
  OuterClass.InnerClass innerObject = outerObject.new InnerClass();
  ```

- If the inner class is static, use the following syntax to create an object for it:

  ```
  OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
  ```

A simple use of inner classes is to combine dependent classes into a primary class. This reduces the number of source files. It also makes class files easy to organize since they are all named with the primary class as the prefix. For example, rather than creating the two source files **Test.java** and **A.java** as shown in Figure 15.7a, you can merge class `A` into class `Test` and create just one source file, **Test.java** as shown in Figure 15.7b. The resulting class files are **Test.class** and **Test$A.class**.

Another practical use of inner classes is to avoid class-naming conflicts. Two versions of `A` are defined in Figure 15.7a and 15.7b. You can define them as inner classes to avoid a conflict.

A handler class is designed specifically to create a handler object for a GUI component (e.g., a button). The handler class will not be shared by other applications and therefore is appropriate to be defined inside the main class as an inner class.

Check Point

15.4.1 Can an inner class be used in a class other than the class in which it nests?

15.4.2 Can the modifiers `public`, `protected`, `private`, and `static` be used for inner classes?

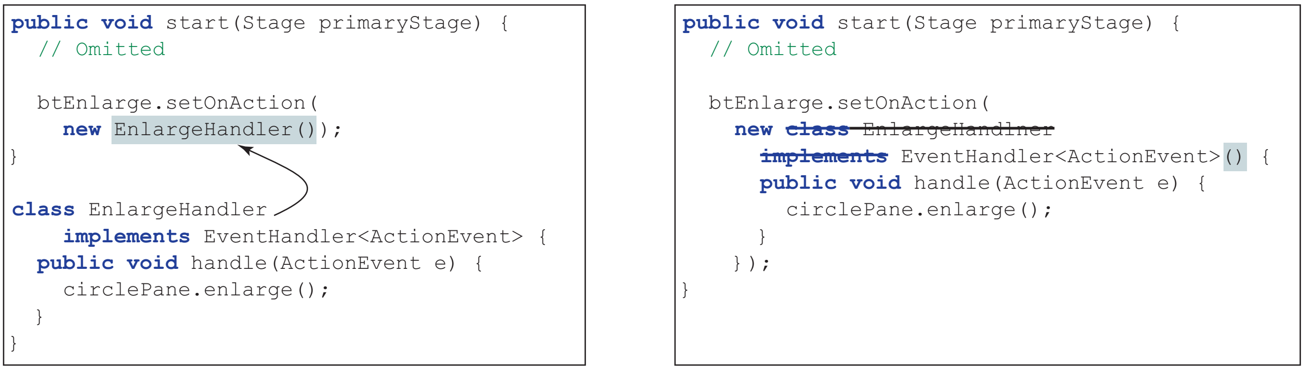
15.5 Anonymous Inner-Class Handlers

Key Point

An anonymous inner class is an inner class without a name. It combines defining an inner class and creating an instance of the class into one step.

anonymous inner class

Inner-class handlers can be shortened using *anonymous inner classes*. The inner class in Listing 15.3 can be replaced by an anonymous inner class as shown below. The complete code is available at liveexample.pearsoncmg.com/html/ControlCircleWithAnonymousInnerClass.html.



```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new EnlargeHandler());
}

class EnlargeHandler
    implements EventHandler<ActionEvent> {
  public void handle(ActionEvent e) {
    circlePane.enlarge();
  }
}
```

(a) Inner class `Enlargelistener`

```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new ~~class EnlargeHandlner~~
      ~~implements~~ EventHandler<ActionEvent>() {
      public void handle(ActionEvent e) {
        circlePane.enlarge();
      }
    });
}
```

(b) Anonymous inner class

The syntax for an anonymous inner class is shown below.

```
new SuperClassName/InterfaceName() {
  // Implement or override methods in superclass or interface

  // Other methods if necessary
}
```

Since an anonymous inner class is a special kind of inner class, it is treated like an inner class with the following features:

- An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit `extends` or `implements` clause.
- An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
- An anonymous inner class always uses the no-arg constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is `Object()`.
- An anonymous inner class is compiled into a class named `OuterClassName$n.class`. For example, if the outer class `Test` has two anonymous inner classes, they are compiled into `Test$1.class` and `Test$2.class`.

Listing 15.4 gives an example that displays a text and uses four buttons to move a text up, down, left, and right, as shown in Figure 15.8.

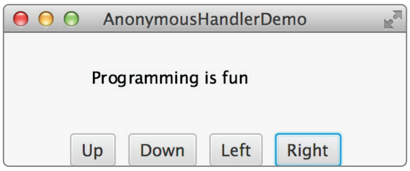


FIGURE 15.8 The program handles the events from four buttons.

LISTING 15.4 AnonymousHandlerDemo.java

VideoNote
Anonymous handler

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
```

anonymous handler

handle event

```
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class AnonymousHandlerDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    Text text = new Text(40, 40, "Programming is fun");
    Pane pane = new Pane(text);

    // Hold four buttons in an HBox
    Button btUp = new Button("Up");
    Button btDown = new Button("Down");
    Button btLeft = new Button("Left");
    Button btRight = new Button("Right");
    HBox hBox = new HBox(btUp, btDown, btLeft, btRight);
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);

    BorderPane borderPane = new BorderPane(pane);
    borderPane.setBottom(hBox);

    // Create and register the handler
    btUp.setOnAction(new EventHandler<ActionEvent>() {
      @Override // Override the handle method
      public void handle(ActionEvent e) {
        text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
      }
    });

    btDown.setOnAction(new EventHandler<ActionEvent>() {
      @Override // Override the handle method
      public void handle(ActionEvent e) {
        text.setY(text.getY() < pane.getHeight() ?
          text.getY() + 5 : pane.getHeight());
      }
    });

    btLeft.setOnAction(new EventHandler<ActionEvent>() {
      @Override // Override the handle method
      public void handle(ActionEvent e) {
        text.setX(text.getX() > 0 ? text.getX() - 5 : 0);
      }
    });

    btRight.setOnAction(new EventHandler<ActionEvent>() {
      @Override // Override the handle method
      public void handle(ActionEvent e) {
        text.setX(text.getX() < pane.getWidth() - 100?
          text.getX() + 5 : pane.getWidth() - 100);
      }
    });

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 400, 350);
    primaryStage.setTitle("AnonymousHandlerDemo"); // Set title
65       primaryStage.setScene(scene); // Place the scene in the stage
66       primaryStage.show(); // Display the stage
67     }
68   }
```

The program creates four handlers using anonymous inner classes (lines 32–60). Without using anonymous inner classes, you would have to create four separate classes. An anonymous handler works the same way as that of an inner-class handler. The program is condensed using an anonymous inner class. Another benefit of using anonymous inner class is that the handler can access local variables. In this example, the event handler references local variable `text` (lines 35, 42, 50, and 57).

The anonymous inner classes in this example are compiled into `AnonymousHandlerDemo$1.class`, `AnonymousHandlerDemo$2.class`, `AnonymousHandlerDemo$3.class`, and `AnonymousHandlerDemo$4.class`.

Check Point

15.5.1 If class `A` is an inner class in class `B`, what is the .class file for `A`? If class `B` contains two anonymous inner classes, what are the .class file names for these two classes?

15.5.2 What is wrong in the following code?

```
public class Test extends Application {
  public void start(Stage stage) {
    Button btOK = new Button("OK");
  }

  private class Handler implements
      EventHandler<ActionEvent> {
    public void handle(Action e) {
      System.out.println(e.getSource());
    }
  }
}
```

(a)

```
public class Test extends Application {
  public void start(Stage stage) {
    Button btOK = new Button("OK");

    btOK.setOnAction(
      new EventHandler<ActionEvent> {
        public void handle
            (ActionEvent e) {
          System.out.println
            (e.getSource());
      }
    }  // Something missing here
   }
}
```

(b)

15.6 Simplifying Event Handling Using Lambda Expressions

Lambda expressions can be used to greatly simplify coding for event handling.

Key Point

lambda expression

Lambda expression is a new feature in Java 8. Lambda expressions can be viewed as an anonymous class with a concise syntax. For example, the following code in (a) can be greatly simplified using a lambda expression in (b) in three lines. Note that the interface `EventHandler<ActionEvent>` and the method `handle` in (a) are removed in (b). This simplification is possible because that the Java compiler can automatically infer that the `setOnAction` method requires an instance of `EvenHandler<ActionEvent>` and the handle is the only method in the `EvenHandler<ActionEvent>` interface. The complete code that contains the lambda expression in (b) can be seen at liveexample.pearsoncmg.com/html/ControlCircleWithLambdaExpression.html.

```
btEnlarge.setOnAction {
  new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
      // Code for processing event e
    }
 }
});
```

(a) Anonymous inner class event handler

```
btEnlarge.setOnAction(e -> {
  // Code for processing event e
});
```

(b) Lambda expression event handler

The basic syntax for a lambda expression is either

```
(type1 param1, type2 param2, . . . ) -> expression
```

or

```
(type1 param1, type2 param2, . . . ) -> { statements; }
```

The data type for a parameter may be explicitly declared or implicitly inferred by the compiler. The parentheses can be omitted if there is only one parameter without an explicit data type. The curly braces can be omitted if there is only one statement. For example, the following lambda expressions are all equivalent. *Note there is no semicolon after the statement in (d).*

```
(ActiionEvent e) -> {
  circlePane.enlarge(); }
```

(a) Lambda expression with one statement

```
(e) -> {
  circlePane.enlarge(); }
```

(b) Omit parameter data type

```
e -> {
  circlePane.enlarge(); }
```

(c) Omit parentheses

```
e ->
  circlePane.enlarge()
```

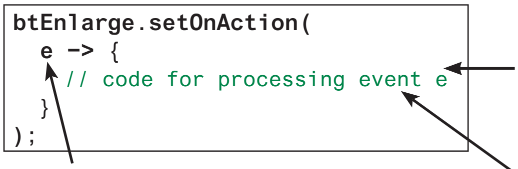
(d) Omit braces

The compiler treats a lambda expression as if it is an object created from an anonymous inner class. The compiler processes a lambda expression in three steps: (1) identify the lambda expression type, (2) identify the parameter types, and (3) identify statements. Consider the following lambda expression:

```
btEnlarge.setOnAction(
  e -> {
  // Code for processing event e
  }
);
```

It is processed as follows:

Step 1: The compiler recognizes that the object must be an instance of `EventHandler<ActionEvent>`, since the expression is an argument of the `setOnAction` method as shown in the following figure:



```
btEnlarge.setOnAction(
  e -> {
   // code for processing event e
  }
);
```

(1) The compiler recognizes that the lambda expression is an object of the `EventHandler<ActionEvent>` type, because the expression is an argument in the `setOnAction` method.

(2) The compiler recognizes that e is a parameter of the `ActionEvent` type, since the `EventHandler<ActionEvent>` interface defines the handle method with a parameter of the `ActionEvent` type.

(3) The compiler recognizes that the code for processing event `e` are the statements in the handle method.

Step 2: Since the `EventHandler` interface defines the `handle` method with a parameter of the `ActionEvent` type, the compiler recognizes that `e` is a parameter of the `ActionEvent` type.

Step 3: The compiler recognizes that the code for processing `e` is the statements in the body of the `handle` method.

The `EventHandler` interface contains just one method named `handle`. The statements in the lambda expression are all for that method. If it contains multiple methods, the compiler will not be able to compile the lambda expression. Therefore, for the compiler to understand lambda expressions, the interface must contain exactly one abstract method. Such an interface is known as a *Single Abstract Method* (SAM) *interface*.

SAM interface

In essence, a lambda expression creates an object and the object performs a function by invoking this single method. Thus, a SAM interface is also known as a *functional interface*, and an instance of a functional interface is known as a *function object*. Since a lambda expression is squarely on defining a function, a lambda expression is also called a *lambda function*. The terms lambda expression and lambda function are interchangeable.

functional interface
function object
lambda function
functional programming

Listing 15.4 can be simplified using lambda expressions as shown in Listing 15.5.

Listing 15.5 LambdaHandlerDemo.java

```java
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class LambdaHandlerDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    Text text = new Text(40, 40, "Programming is fun");
    Pane pane = new Pane(text);

    // Hold four buttons in an HBox
    Button btUp = new Button("Up");
    Button btDown = new Button("Down");
    Button btLeft = new Button("Left");
    Button btRight = new Button("Right");
    HBox hBox = new HBox(btUp, btDown, btLeft, btRight);
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);

    BorderPane borderPane = new BorderPane(pane);
    borderPane.setBottom(hBox);

    // Create and register the handler
    btUp.setOnAction((ActionEvent e) -> {
      text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
    });

    btDown.setOnAction((e) -> {
      text.setY(text.getY() < pane.getHeight() ?
        text.getY() + 5 : pane.getHeight());
    });

    btLeft.setOnAction(e -> {
      text.setX(text.getX() > 0 ? text.getX() - 5 : 0);
    });

    btRight.setOnAction(e ->
      text.setX(text.getX() < pane.getWidth() - 100?
        text.getX() + 5 : pane.getWidth() - 100)
    );
```

lambda handler

lambda handler

lambda handler

lambda handler

```
49
50      // Create a scene and place it in the stage
51      Scene scene = new Scene(borderPane, 400, 350);
52      primaryStage.setTitle("AnonymousHandlerDemo"); // Set title
53      primaryStage.setScene(scene); // Place the scene in the stage
54      primaryStage.show(); // Display the stage
55    }
56  }
```

The program creates four handlers using lambda expressions (lines 32–48). Using lambda expressions, the code is shorter and cleaner. As seen in this example, lambda expressions may have many variations. Line 32 uses a declared type. Line 36 uses an inferred type since the type can be determined by the compiler. Line 41 omits the parentheses for a single inferred type. Line 45 omits the braces for a single statement in the body.

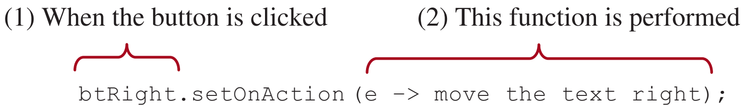
inner class, anonymous class, or Lambda?

You can handle events by defining handler classes using inner classes, anonymous inner classes, or lambda expressions. We recommend you use lambda expressions because it produces a shorter, clearer, and cleaner code.

simplify syntax

simplify concept

Using lambda expressions not only simplifies the syntax, but also simplifies the event-handling concept. For the statement in line 45,



(1) When the button is clicked (2) This function is performed

`btRight.setOnAction(e -> move the text right);`

you can now simply say that when the **`btRight`** button is clicked, the lambda function is invoked to move the text right.

You can define a custom functional interface and use it in a lambda expression. Consider the following example in Listing 15.6:

LISTING 15.6 TestLambda.java

```
public class TestLambda {
  public static void main(String[] args) {
    TestLambda test = new TestLambda();
    test.setAction1(() -> System.out.print("Action 1! "));
    test.setAction2(e -> System.out.print(e + " "));
    System.out.println(test.getValue((e1, e2) -> e1 + e2));
  }

  public void setAction1(T1 t) {
    t.m1();
  }

  public void setAction2(T2 t) {
    t.m2(4.5);
  }

  public int getValue(T3 t) {
    return t.m3(5, 2);
  }
}

@FunctionalInterface
interface T1 {
  public void m1();
}

@FunctionalInterface
interface T2 {
```

```
29    public void m2(Double d);
30  }
31
32  @FunctionalInterface
33  interface T3 {
34    public int m3(int d1, int d2);
35  }
```

The annotation `@FunctionalInterface` tells the compiler that the interface is a functional interface. Since `T1`, `T2`, and `T3` are all functional interfaces, a lambda expression can be used with the methods `setAction1(T1)`, `setAction2(T2)`, and `getValue(T3)`. The statement in line 4 is equivalent to using an anonymous inner class, as follows:

```
test.setAction1(new T1() {
  @Override
  public void m1() {
    System.out.print("Action 1! ");
  }
});
```

15.6.1 What is a lambda expression? What is the benefit of using lambda expressions for event handling? What is the syntax of a lambda expression?

15.6.2 What is a functional interface? Why is a functional interface required for a lambda expression?

15.6.3 Replace the code in lines 5 and 6 in TestLambda.java using anonymous inner classes.

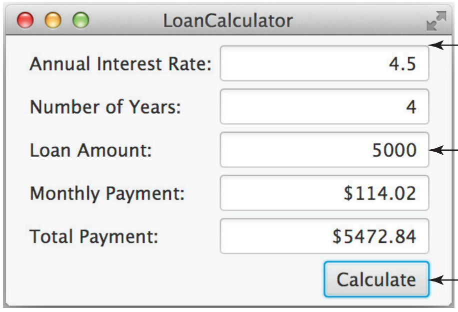
15.7 Case Study: Loan Calculator

This case study develops a loan calculator using event-driven programming with GUI controls.

Now, we will write the program for the loan-calculator problem presented at the beginning of this chapter. Here are the major steps in the program:

1. Create the user interface, as shown in Figure 15.9.
    a. Create a `GridPane`. Add labels, text fields, and button to the pane.
    b. Set the alignment of the button to the right.
2. Process the event.

    Create and register the handler for processing the button-clicking action event. The handler obtains the user input on the loan amount, interest rate, and number of years, computes the monthly and total payments, and displays the values in the text fields.



Annual Interest Rate: 4.5
Number of Years: 4
Loan Amount: 5000
Monthly Payment: $114.02
Total Payment: $5472.84

`GridPane`

Text field is right aligned

Button is right aligned

Figure 15.9 The program computes loan payments.

The complete program is given in Listing 15.7.

LISTING 15.7 LoanCalculator.java

```java
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.geometry.HPos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class LoanCalculator extends Application {
  private TextField tfAnnualInterestRate = new TextField();
  private TextField tfNumberOfYears = new TextField();
  private TextField tfLoanAmount = new TextField();
  private TextField tfMonthlyPayment = new TextField();
  private TextField tfTotalPayment = new TextField();
  private Button btCalculate = new Button("Calculate");

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create UI
    GridPane gridPane = new GridPane();
    gridPane.setHgap(5);
    gridPane.setVgap(5);
    gridPane.add(new Label("Annual Interest Rate:"), 0, 0);
    gridPane.add(tfAnnualInterestRate, 1, 0);
    gridPane.add(new Label("Number of Years:"), 0, 1);
    gridPane.add(tfNumberOfYears, 1, 1);
    gridPane.add(new Label("Loan Amount:"), 0, 2);
    gridPane.add(tfLoanAmount, 1, 2);
    gridPane.add(new Label("Monthly Payment:"), 0, 3);
    gridPane.add(tfMonthlyPayment, 1, 3);
    gridPane.add(new Label("Total Payment:"), 0, 4);
    gridPane.add(tfTotalPayment, 1, 4);
    gridPane.add(btCalculate, 1, 5);

    // Set properties for UI
    gridPane.setAlignment(Pos.CENTER);
    tfAnnualInterestRate.setAlignment(Pos.BOTTOM_RIGHT);
    tfNumberOfYears.setAlignment(Pos.BOTTOM_RIGHT);
    tfLoanAmount.setAlignment(Pos.BOTTOM_RIGHT);
    tfMonthlyPayment.setAlignment(Pos.BOTTOM_RIGHT);
    tfTotalPayment.setAlignment(Pos.BOTTOM_RIGHT);
    tfMonthlyPayment.setEditable(false);
    tfTotalPayment.setEditable(false);
    GridPane.setHalignment(btCalculate, HPos.RIGHT);

    // Process events
    btCalculate.setOnAction(e -> calculateLoanPayment());

    // Create a scene and place it in the stage
    Scene scene = new Scene(gridPane, 400, 250);
    primaryStage.setTitle("LoanCalculator"); // Set title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
```

text fields

button

create a grid pane

add to grid pane

register handler

```
57
58    private void calculateLoanPayment() {
59      // Get values from text fields
60      double interest =
61        Double.parseDouble(tfAnnualInterestRate.getText());
62      int year = Integer.parseInt(tfNumberOfYears.getText());
63      double loanAmount =
64        Double.parseDouble(tfLoanAmount.getText());
65
66      // Create a loan object. Loan defined in Listing 10.2
67      Loan loan = new Loan(interest, year, loanAmount);
68
69      // Display monthly payment and total payment
70      tfMonthlyPayment.setText(String.format("$%.2f",
71        loan.getMonthlyPayment()));
72      tfTotalPayment.setText(String.format("$%.2f",
73        loan.getTotalPayment()));
74    }
75  }
```

get input

create loan

set result

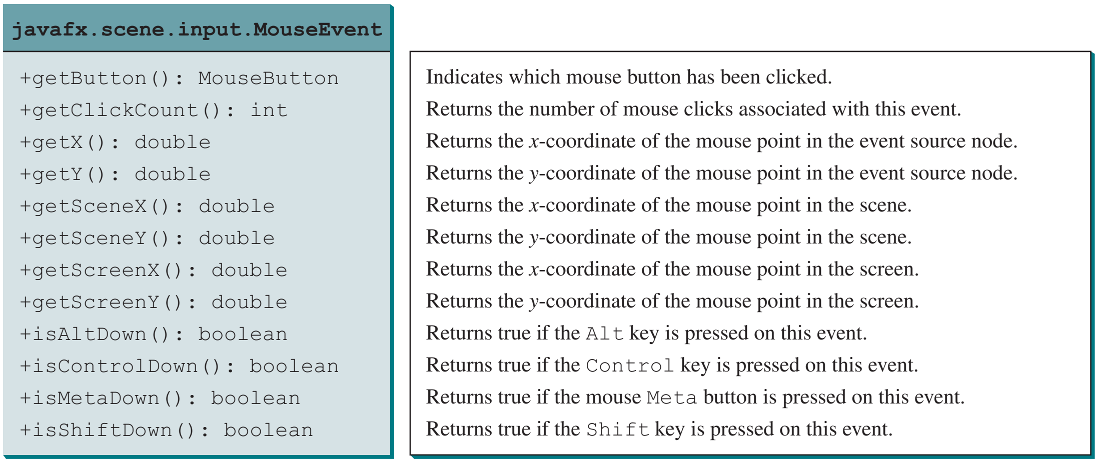
The user interface is created in the **`start`** method (lines 22–46). The button is the source of the event. A handler is created and registered with the button (line 49). The button handler invokes the **`calculateLoanPayment()`** method to get the interest rate (line 60), number of years (line 62), and loan amount (line 64). Invoking **`tfAnnualInterestRate.getText()`** returns the string text in the **`tfAnnualInterestRate`** text field. The **`Loan`** class is used for computing the loan payments. This class was introduced in Listing 10.2, Loan.java. Invoking **`loan.getMonthlyPayment()`** returns the monthly payment for the loan (line 71). The **`String.format`** method, introduced in Section 10.10.7, is used to format a number into a desirable format and returns it as a string (lines 70 and 72). Invoking the **`setText`** method on a text field sets a string value in the text field.

15.8 Mouse Events

*A **`MouseEvent`** is fired whenever a mouse button is pressed, released, clicked, moved, or dragged on a node or a scene.*

Key Point

The **`MouseEvent`** object captures the event, such as the number of clicks associated with it, the location (the *x*- and *y*-coordinates) of the mouse, or which mouse button was pressed, as shown in Figure 15.10.



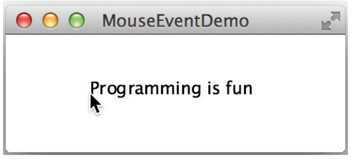
`javafx.scene.input.MouseEvent`	
`+getButton(): MouseButton`	Indicates which mouse button has been clicked.
`+getClickCount(): int`	Returns the number of mouse clicks associated with this event.
`+getX(): double`	Returns the *x*-coordinate of the mouse point in the event source node.
`+getY(): double`	Returns the *y*-coordinate of the mouse point in the event source node.
`+getSceneX(): double`	Returns the *x*-coordinate of the mouse point in the scene.
`+getSceneY(): double`	Returns the *y*-coordinate of the mouse point in the scene.
`+getScreenX(): double`	Returns the *x*-coordinate of the mouse point in the screen.
`+getScreenY(): double`	Returns the *y*-coordinate of the mouse point in the screen.
`+isAltDown(): boolean`	Returns true if the `Alt` key is pressed on this event.
`+isControlDown(): boolean`	Returns true if the `Control` key is pressed on this event.
`+isMetaDown(): boolean`	Returns true if the mouse `Meta` button is pressed on this event.
`+isShiftDown(): boolean`	Returns true if the `Shift` key is pressed on this event.

FIGURE 15.10 The **`MouseEvent`** class encapsulates information for mouse events.

Four constants—`PRIMARY`, `SECONDARY`, `MIDDLE`, and `NONE`—are defined in `MouseButton` to indicate the left, right, middle, and none mouse buttons, respectively. You can use the `getButton()` method to detect which button is pressed. For example, `getButton() == MouseButton.SECONDARY` tests if the right button was pressed. You can also use the `isPrimaryButtonDown()`, `isSecondaryButtonDown()`, and `isMiddleButtonDown()` to test if the primary button, second button, or middle button is pressed.

detect mouse buttons

The mouse events and their corresponding registration methods for handlers are listed in Table 15.1. To demonstrate using mouse events, we give an example that displays a message in a pane and enables the message to be moved using a mouse. The message moves as the mouse is dragged, and it is always displayed at the mouse point. Listing 15.8 gives the program. A sample run of the program is shown in Figure 15.11.



Programming is fun

FIGURE 15.11 You can move the message by dragging the mouse.

VideoNote

Move message using the mouse

LISTING 15.8 MouseEventDemo.java

```java
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.layout.Pane;
 4 import javafx.scene.text.Text;
 5 import javafx.stage.Stage;
 6
 7 public class MouseEventDemo extends Application {
 8   @Override // Override the start method in the Application class
 9   public void start(Stage primaryStage) {
10     // Create a pane and set its properties
11     Pane pane = new Pane();
12     Text text = new Text(20, 20, "Programming is fun");
13     pane.getChildren().addAll(text);
14     text.setOnMouseDragged(e -> {
15       text.setX(e.getX());
16       text.setY(e.getY());
17     });
18
19     // Create a scene and place it in the stage
20     Scene scene = new Scene(pane, 300, 100);
21     primaryStage.setTitle("MouseEventDemo"); // Set the stage title
22     primaryStage.setScene(scene); // Place the scene in the stage
23     primaryStage.show(); // Display the stage
24   }
25 }
```

create a pane
create a text
add text to a pane
lambda handler
reset text position

Each node or scene can fire mouse events. The program creates a `Text` (line 12) and registers a handler to handle move dragged event (line 14). Whenever a mouse is dragged, the text's *x*- and *y*-coordinates are set to the mouse position (lines 15 and 16).

Check Point

15.8.1 What method do you use to get the mouse-point position for a mouse event?

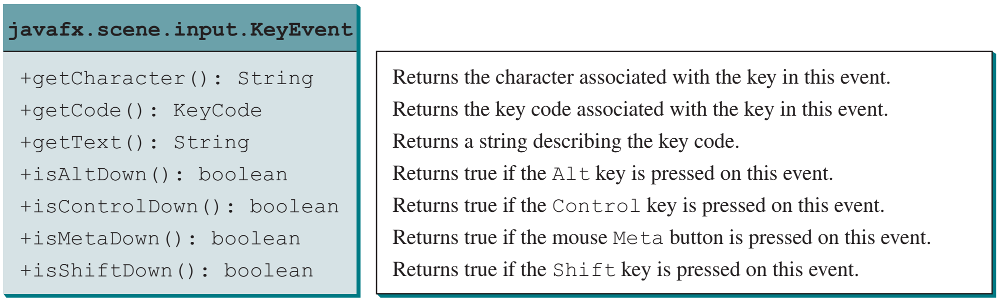
15.8.2 What methods do you use to register a handler for mouse-pressed, -released, -clicked, -entered, -exited, -moved, and -dragged events?

15.9 Key Events

Key Point

A **`KeyEvent`** *is fired whenever a key is pressed, released, or typed on a node or a scene.*

Key events enable the use of the keys to control and perform actions, or get input from the keyboard. The **`KeyEvent`** object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key, as shown in Figure 15.12.



`javafx.scene.input.KeyEvent`	
`+getCharacter(): String`	Returns the character associated with the key in this event.
`+getCode(): KeyCode`	Returns the key code associated with the key in this event.
`+getText(): String`	Returns a string describing the key code.
`+isAltDown(): boolean`	Returns true if the `Alt` key is pressed on this event.
`+isControlDown(): boolean`	Returns true if the `Control` key is pressed on this event.
`+isMetaDown(): boolean`	Returns true if the mouse `Meta` button is pressed on this event.
`+isShiftDown(): boolean`	Returns true if the `Shift` key is pressed on this event.

FIGURE 15.12 The **`KeyEvent`** class encapsulates information about key events.

The key events key pressed, key released, and key typed and their corresponding registration methods for handlers are listed in Table 15.1. The key pressed handler is invoked when a key is pressed, the key released handler is invoked when a key is released, and the key typed handler is invoked when a Unicode character is entered. If a key does not have a Unicode (e.g., function keys, modifier keys, action keys, arrow keys, and control keys), the key typed handler will not be invoked.

key code

Every key event has an associated code that is returned by the **`getCode()`** method in **`KeyEvent`**. The *key codes* are constants defined in **`KeyCode`**. Table 15.2 lists some constants. **`KeyCode`** is an **`enum`** type. For use of **`enum`** types, see Appendix I. For the key-pressed and key-released events, **`getCode()`** returns the value as defined in the table, **`getText()`** returns a string that describes the key code, and **`getCharacter()`** returns an empty string. For the key-typed event, **`getCode()`** returns **`UNDEFINED`** and **`getCharacter()`** returns the Unicode character or a sequence of characters associated with the key-typed event.

TABLE 15.2 KeyCode Constants

Constant	*Description*	*Constant*	*Description*
`HOME`	The Home key	**`CONTROL`**	The Control key
`END`	The End key	**`SHIFT`**	The Shift key
`PAGE_UP`	The Page Up key	**`BACK_SPACE`**	The Backspace key
`PAGE_DOWN`	The Page Down key	**`CAPS`**	The Caps Lock key
`UP`	The up-arrow key	**`NUM_LOCK`**	The Num Lock key
`DOWN`	The down-arrow key	**`ENTER`**	The Enter key
`LEFT`	The left-arrow key	**`UNDEFINED`**	The **`keyCode`** unknown
`RIGHT`	The right-arrow key	**`F1`** to **`F12`**	The function keys from F1 to F12
`ESCAPE`	The Esc key	**`0`** to **`9`**	The number keys from 0 to 9
`TAB`	The Tab key	**`A`** to **`Z`**	The letter keys from A to Z

The program in Listing 15.9 displays a user-input character. The user can move the character up, down, left, and right, using the up-, down-, left-, and right-arrow keys, respectively. Figure 15.13 contains a sample run of the program.

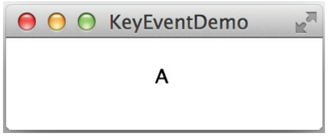


FIGURE 15.13 The program responds to key events by displaying a character and moving it up, down, left, or right.

LISTING 15.9 KeyEventDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.layout.Pane;
 4 import javafx.scene.text.Text;
 5 import javafx.stage.Stage;
 6
 7 public class KeyEventDemo extends Application {
 8   @Override // Override the start method in the Application class
 9   public void start(Stage primaryStage) {
10     // Create a pane and set its properties
11     Pane pane = new Pane();
12     Text text = new Text(20, 20, "A");
13
14     pane.getChildren().add(text);
15     text.setOnKeyPressed(e -> {
16       switch (e.getCode()) {
17         case DOWN: text.setY(text.getY() + 10); break;
18         case UP: text.setY(text.getY() - 10); break;
19         case LEFT: text.setX(text.getX() - 10); break;
20         case RIGHT: text.setX(text.getX() + 10); break;
21         default:
22           if (e.getText().length() > 0)
23             text.setText(e.getText());
24       }
25     });
26
27     // Create a scene and place it in the stage
28     Scene scene = new Scene(pane);
29     primaryStage.setTitle("KeyEventDemo"); // Set the stage title
30     primaryStage.setScene(scene); // Place the scene in the stage
31     primaryStage.show(); // Display the stage
32
33     text.requestFocus(); // text is focused to receive key input
34   }
35 }
```

create a pane

register handler

get the key pressed

move a character

set a new character

request focus on text

The program creates a pane (line 11), creates a text (line 12), and places the text into the pane (line 14). The text registers the handler for the key-pressed event in lines 15–25. When a key is pressed, the handler is invoked. The program uses `e.getCode()` (line 16) to obtain the key code and `e.getText()` (line 23) to get the character for the key. Note for a nonprintable character such as a CTRL key or SHIFT key, `e.getText()` returns an empty string. When a non-arrow key is pressed, the character is displayed (lines 22 and 23). When an arrow key is pressed, the character moves in the direction indicated by the arrow key (lines 17–20). Note in a switch statement for an enum-type value, the cases are for the enum constants (lines 16–24). The constants are unqualified. For example, using `KeyCode.DOWN` in the case clause would be wrong (see Appendix I).

`requestFocus()`

Only a focused node can receive `KeyEvent`. Invoking `requestFocus()` on `text` enables `text` to receive key input (line 33). This method must be invoked after the stage is displayed. The program would work fine if `text` is replaced by `scene` in line 15 as follows:

```
scene.setOnKeyPressed(e -> { ... });
```

You don't need to invoke `scene.requestFocus()` because scene is a top-level container for receiving key events.

We can now add more control for our `ControlCircle` example in Listing 15.3 to increase/decrease the circle radius by clicking the left/right mouse button or by pressing the up and down arrow keys. The new program is given in Listing 15.10.

Listing 15.10 ControlCircleWithMouseAndKey.java

```
 1  import javafx.application.Application;
 2  import javafx.geometry.Pos;
 3  import javafx.scene.Scene;
 4  import javafx.scene.control.Button;
 5  import javafx.scene.input.KeyCode;
 6  import javafx.scene.input.MouseButton;
 7  import javafx.scene.layout.HBox;
 8  import javafx.scene.layout.BorderPane;
 9  import javafx.stage.Stage;
10
11  public class ControlCircleWithMouseAndKey extends Application {
12    private CirclePane circlePane = new CirclePane();
13
14    @Override // Override the start method in the Application class
15    public void start(Stage primaryStage) {
16      // Hold two buttons in an HBox
17      HBox hBox = new HBox();
18      hBox.setSpacing(10);
19      hBox.setAlignment(Pos.CENTER);
20      Button btEnlarge = new Button("Enlarge");
21      Button btShrink = new Button("Shrink");
22      hBox.getChildren().add(btEnlarge);
23      hBox.getChildren().add(btShrink);
24
25      // Create and register the handler
26      btEnlarge.setOnAction(e -> circlePane.enlarge());
27      btShrink.setOnAction(e -> circlePane.shrink());
28
29      BorderPane borderPane = new BorderPane();
30      borderPane.setCenter(circlePane);
31      borderPane.setBottom(hBox);
32      BorderPane.setAlignment(hBox, Pos.CENTER);
33
34      // Create a scene and place it in the stage
35      Scene scene = new Scene(borderPane, 200, 150);
36      primaryStage.setTitle("ControlCircle"); // Set the stage title
37      primaryStage.setScene(scene); // Place the scene in the stage
38      primaryStage.show(); // Display the stage
39
40      circlePane.setOnMouseClicked(e -> {
41        if (e.getButton() == MouseButton.PRIMARY) {
42          circlePane.enlarge();
43        }
```

button handler

mouse-click handler

key-pressed handler

Up-arrow key pressed

Down-arrow key pressed

```
44        else if (e.getButton() == MouseButton.SECONDARY) {
45          circlePane.shrink();
46        }
47      });
48
49      scene.setOnKeyPressed(e -> {
50        if (e.getCode() == KeyCode.UP) {
51          circlePane.enlarge();
52        }
53        else if (e.getCode() == KeyCode.DOWN) {
54          circlePane.shrink();
55        }
56      });
57    }
58  }
```

The **`CirclePane`** class (line 12) is already defined in Listing 15.3 and can be reused in this program.

mouse-clicked event

A handler for mouse-clicked events is created in lines 40–47. If the left mouse button is clicked, the circle is enlarged (lines 41–43); if the right mouse button is clicked, the circle is shrunk (lines 44–46).

key-pressed event

A handler for key-pressed events is created in lines 49–56. If the up arrow key is pressed, the circle is enlarged (lines 50–52); if the down arrow key is pressed, the circle is shrunk (lines 53–55).

Check Point

15.9.1 What methods do you use to register handlers for key-pressed, key-released, and key-typed events? In which classes are these methods defined? (See Table 15.1.)

15.9.2 What method do you use to get the key character for a key-typed event? What method do you use to get the key code for a key-pressed or key-released event?

15.9.3 How do you set focus on a node so it can listen for key events?

15.9.4 If the following code is inserted in line 57 in Listing 15.9, what is the output if the user presses the key for letter A? What is the output if the user presses the up arrow key?

```
circlePane.setOnKeyPressed(e ->
  System.out.println("Key pressed " + e.getCode()));
circlePane.setOnKeyTyped(e ->
  System.out.println("Key typed " + e.getCode()));
```

15.10 Listeners for Observable Objects

Key Point

You can add a listener to process a value change in an observable object.

observable object

An instance of **`Observable`** is known as an *observable object*, which contains the **`addListener(InvalidationListener listener)`** method for adding a listener. The listener class must implement the functional interface **`InvalidationListener`** to override the **`invalidated(Observable o)`** method for handling the value change. Once the value is changed in the **`Observable`** object, the listener is notified by invoking its **`invalidated(Observable o)`** method. Every binding property is an instance of **`Observable`**. Listing 15.11 gives an example of observing and handling a change in a **`DoubleProperty`** object **`balance`**.

LISTING 15.11 `ObservablePropertyDemo.java`

```
import javafx.beans.InvalidationListener;
import javafx.beans.Observable;
import javafx.beans.property.DoubleProperty;
```

```
 4  import javafx.beans.property.SimpleDoubleProperty;
 5
 6  public class ObservablePropertyDemo {
 7    public static void main(String[] args) {
 8      DoubleProperty balance = new SimpleDoubleProperty();
 9      balance.addListener(new InvalidationListener() {
10        public void invalidated(Observable ov) {
11          System.out.println("The new value is " +
12            balance.doubleValue());
13        }
14      });
15
16      balance.set(4.5);
17    }
18  }
```

observable property
add listener
handle change

```
The new value is 4.5
```

When line 16 is executed, it causes a change in balance, which notifies the listener by invoking the listener's `invalidated` method.

Note the anonymous inner class in lines 9–14 can be simplified using a lambda expression as follows:

```
balance.addListener(ov -> {
  System.out.println("The new value is " +
    balance.doubleValue());
});
```

Listing 15.12 gives a program that displays a circle with its bounding rectangle, as shown in Figure 15.14. The circle and rectangle are automatically resized when the user resizes the window.

LISTING 15.12 ResizableCircleRectangle.java

```
import javafx.application.Application;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;

public class ResizableCircleRectangle extends Application {
  // Create a circle and a rectangle
  private Circle circle = new Circle(60);
  private Rectangle rectangle = new Rectangle(120, 120);

  // Place clock and label in border pane
  private StackPane pane = new StackPane();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    circle.setFill(Color.GRAY);
    rectangle.setFill(Color.WHITE);
    rectangle.setStroke(Color.BLACK);
    pane.getChildren().addAll(rectangle, circle);
```

set a new width for clock
set a new height for clock

```
24
25       // Create a scene and place the pane in the stage
26       Scene scene = new Scene(pane, 140, 140);
27       primaryStage.setTitle("ResizableCircleRectangle");
28       primaryStage.setScene(scene); // Place the scene in the stage
29       primaryStage.show(); // Display the stage
30
31       pane.widthProperty().addListener(ov -> resize());
32       pane.heightProperty().addListener(ov -> resize());
33     }
34
35     private void resize() {
36       double length = Math.min(pane.getWidth(), pane.getHeight());
37       circle.setRadius(length / 2 - 15);
38       rectangle.setWidth(length - 30);
39       rectangle.setHeight(length - 30);
40     }
41   }
```

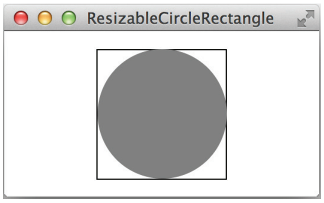


Figure 15.14 The program places a rectangle and a circle inside a stack pane, and automatically sets their sizes when the window is resized.

The program registers the listeners for the stack pane's `width` and `height` properties (lines 31 and 32). When the user resizes the window, the pane's size is changed, so the listeners are called to invoke the `resize()` method to change the size of the circle and rectangle (lines 35–40).

Check Point

15.10.1 What would happen if you replace `pane` with `scene` or `primaryStage` in lines 31–32?

15.11 Animation

Key Point

VideoNote
Animate a rising flag

JavaFX provides the `Animation` *class with the core functionality for all animations.*

Suppose you want to write a program that animates a rising flag, as shown in Figure 15.15. How do you accomplish the task? There are several ways to program this. An effective one is to use the subclasses of the JavaFX `Animation` class, which is the subject of this section.

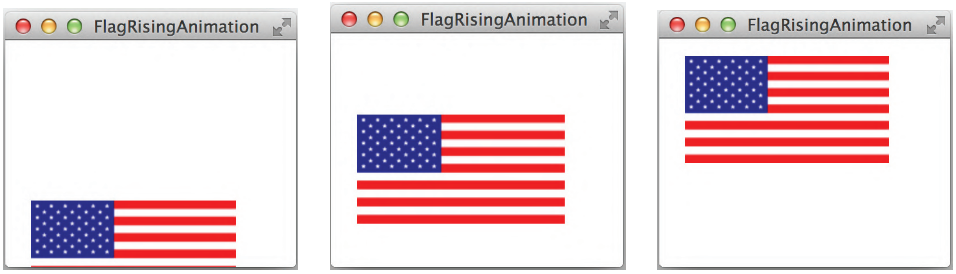
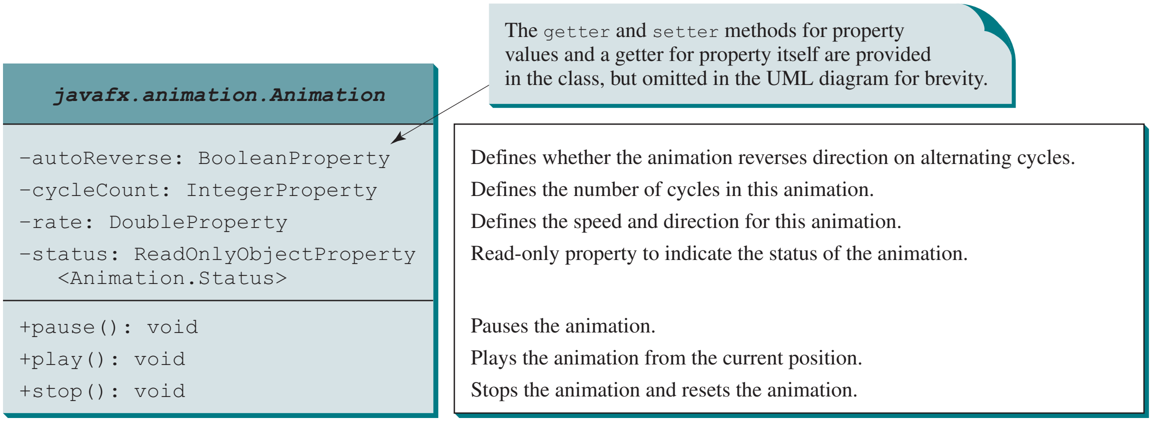


Figure 15.15 The animation simulates a flag rising. *Source*: booka/Fotolia.

The abstract **`Animation`** class provides the core functionalities for animations in JavaFX, as shown in Figure 15.16. Many concrete subclasses of **`Animation`** are provided in JavaFX. This section introduces **`PathTransition`**, **`FadeTransition`**, and **`Timeline`**.



The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

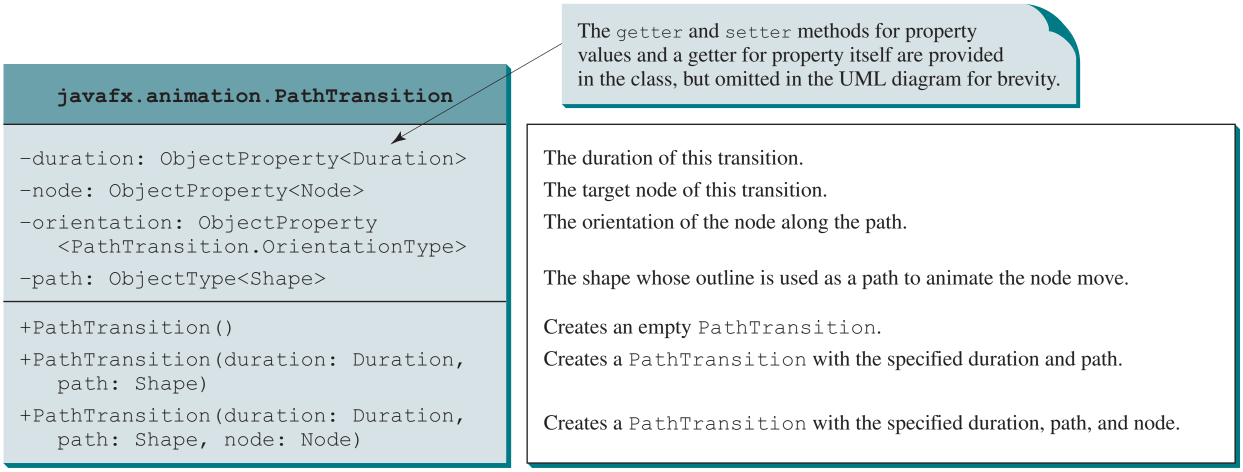
javafx.animation.Animation	
`-autoReverse: BooleanProperty`	Defines whether the animation reverses direction on alternating cycles.
`-cycleCount: IntegerProperty`	Defines the number of cycles in this animation.
`-rate: DoubleProperty`	Defines the speed and direction for this animation.
`-status: ReadOnlyObjectProperty <Animation.Status>`	Read-only property to indicate the status of the animation.
`+pause(): void`	Pauses the animation.
`+play(): void`	Plays the animation from the current position.
`+stop(): void`	Stops the animation and resets the animation.

FIGURE 15.16 The abstract **`Animation`** class is the root class for JavaFX animations.

The **`autoReverse`** is a Boolean property that indicates whether an animation will reverse its direction on the next cycle. The **`cycleCount`** indicates the number of the cycles for the animation. You can use the constant **`Timeline.INDEFINITE`** to indicate an indefinite number of cycles. The **`rate`** defines the speed of the animation. A negative rate value indicates the opposite direction for the animation. The **`status`** is a read-only property that indicates the status of the animation (**`Animation.Status.PAUSED`**, **`Animation.Status.RUNNING`**, and **`Animation.Status.STOPPED`**). The methods **`pause()`**, **`play()`**, and **`stop()`** pause, play, and stop an animation, respectively.

15.11.1 PathTransition

The **`PathTransition`** class animates the moves of a node along a path from one end to the other over a given time. **`PathTransition`** is a subtype of **`Animation`**. The UML class diagram for the class is shown in Figure 15.17.



The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.animation.PathTransition	
`-duration: ObjectProperty<Duration>`	The duration of this transition.
`-node: ObjectProperty<Node>`	The target node of this transition.
`-orientation: ObjectProperty <PathTransition.OrientationType>`	The orientation of the node along the path.
`-path: ObjectType<Shape>`	The shape whose outline is used as a path to animate the node move.
`+PathTransition()`	Creates an empty `PathTransition`.
`+PathTransition(duration: Duration, path: Shape)`	Creates a `PathTransition` with the specified duration and path.
`+PathTransition(duration: Duration, path: Shape, node: Node)`	Creates a `PathTransition` with the specified duration, path, and node.

FIGURE 15.17 The **`PathTransition`** class defines an animation for a node along a path.

The **`Duration`** class defines a duration of time. It is an immutable class. The class defines constants **`INDEFINITE`**, **`ONE`**, **`UNKNOWN`**, and **`ZERO`** to represent an indefinte duration, one millisecond, unknown, and zero duration, respectively. You can use **`new Duration(double millis)`** to create an instance of **`Duration`**, the **`add`**, **`subtract`**, **`multiply`**, and **`divide`** methods to perform arithmetic operations, and the **`toHours()`**, **`toMinutes()`**, **`toSeconds()`**, and **`toMillis()`** to return the number of hours, minutes, seconds, and milliseconds in this duration, respectively. You can also use **`compareTo`** to compare two durations.

The constants **`NONE`** and **`ORTHOGONAL_TO_TANGENT`** are defined in **`PathTransition.OrientationType`**. The latter specifies that the node is kept perpendicular to the path's tangent along the geometric path.

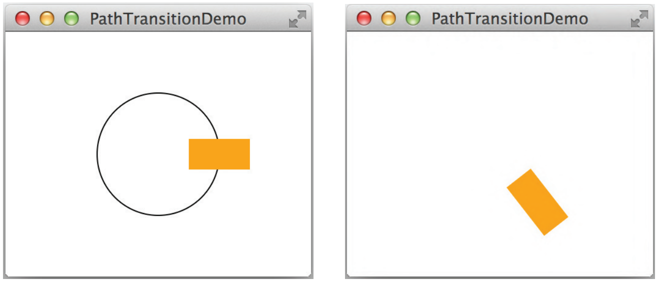
Listing 15.13 gives an example that moves a rectangle along the outline of a circle, as shown in Figure 15.18a.

LISTING 15.13 PathTransitionDemo.java

```java
import javafx.animation.PathTransition;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class PathTransitionDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane
    Pane pane = new Pane();

    // Create a rectangle
    Rectangle rectangle = new Rectangle (0, 0, 25, 50);
    rectangle.setFill(Color.ORANGE);

    // Create a circle
    Circle circle = new Circle(125, 100, 50);
    circle.setFill(Color.WHITE);
    circle.setStroke(Color.BLACK);

    // Add circle and rectangle to the pane
    pane.getChildren().add(circle);
    pane.getChildren().add(rectangle);

    // Create a path transition
    PathTransition pt = new PathTransition();
    pt.setDuration(Duration.millis(4000));
    pt.setPath(circle);
    pt.setNode(rectangle);
    pt.setOrientation(
      PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);
    pt.setCycleCount(Timeline.INDEFINITE);
    pt.setAutoReverse(true);
    pt.play(); // Start animation

    circle.setOnMousePressed(e -> pt.pause());
    circle.setOnMouseReleased(e -> pt.play());
```

create a pane
create a rectangle
create a circle
add circle to pane
add rectangle to pane
create a `PathTransition`
set transition duration
set path in transition
set node in transition
set orientation
set cycle count indefinite
set auto reverse true
play animation
pause animation
resume animation

```
44
45       // Create a scene and place it in the stage
46       Scene scene = new Scene(pane, 250, 200);
47       primaryStage.setTitle("PathTransitionDemo"); // Set the stage title
48       primaryStage.setScene(scene); // Place the scene in the stage
49       primaryStage.show(); // Display the stage
50     }
51   }
```



(a) (b)

FIGURE 15.18 The `PathTransition` animates a rectangle moving along the circle.

The program creates a pane (line 16), a rectangle (line 19), and a circle (line 23). The circle and rectangle are placed in the pane (lines 28 and 29). If the circle was not placed in the pane, you will see the screen shot as shown in Figure 15.18b.

The program creates a path transition (line 32), sets its duration to 4 seconds for one cycle of animation (line 33), sets circle as the path (line 34), sets rectangle as the node (line 35), and sets the orientation to orthogonal to tangent (line 36).

The cycle count is set to indefinite (line 38) so the animation continues forever. The auto reverse is set to true (line 39) so the direction of the move is reversed in the alternating cycle. The program starts animation by invoking the `play()` method (line 40).

If the `pause()` method is replaced by the `stop()` method in line 42, the animation will start over from the beginning when it restarts.

Listing 15.14 gives the program that animates a flag rising, as shown in Figure 15.14.

LISTING 15.14 FlagRisingAnimation.java

```
import javafx.animation.PathTransition;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.ImageView;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Line;
import javafx.stage.Stage;
import javafx.util.Duration;

public class FlagRisingAnimation extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane
    Pane pane = new Pane();

    // Add an image view and add it to pane
    ImageView imageView = new ImageView("image/us.gif");
    pane.getChildren().add(imageView);
```

create a pane

create an image view

add image view to pane

create a path transition

set cycle count

play animation

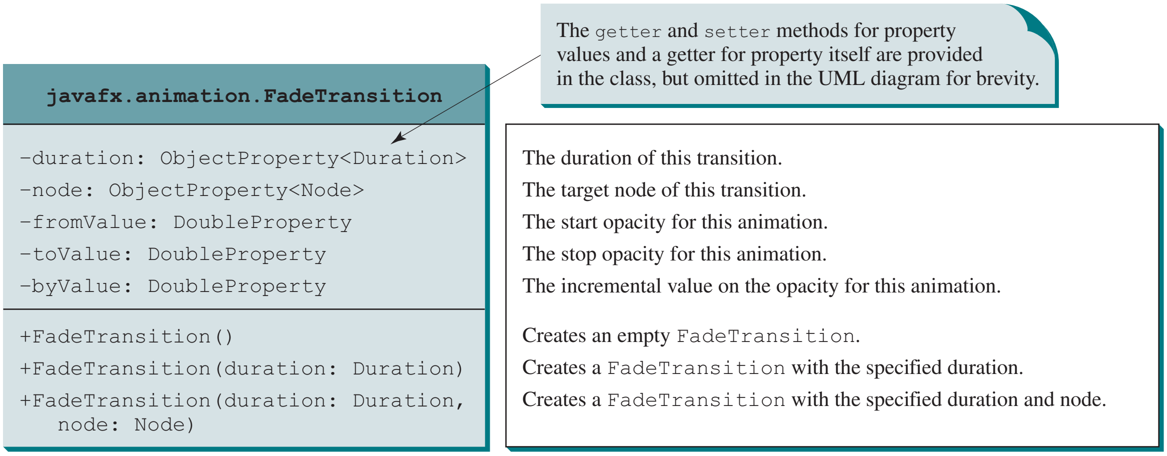
```
19
20      // Create a path transition
21      PathTransition pt = new PathTransition(Duration.millis(10000),
22        new Line(100, 200, 100, 0), imageView);
23      pt.setCycleCount(5);
24      pt.play(); // Start animation
25
26      // Create a scene and place it in the stage
27      Scene scene = new Scene(pane, 250, 200);
28      primaryStage.setTitle("FlagRisingAnimation"); // Set the stage title
29      primaryStage.setScene(scene); // Place the scene in the stage
30      primaryStage.show(); // Display the stage
31    }
32  }
```

The program creates a pane (line 14), an image view from an image file (line 17), and places the image view to the pane (line 18). A path transition is created with a duration of 10 seconds using a line as a path and the image view as the node (lines 21 and 22). The image view will move along the line. Since the line is not placed in the scene, you will not see the line in the window.

The cycle count is set to 5 (line 23) so the animation is repeated five times.

15.11.2 FadeTransition

The **`FadeTransition`** class animates the change of the opacity in a node over a given time. **`FadeTransition`** is a subtype of **`Animation`**. The UML class diagram for the class is shown in Figure 15.19.



The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.animation.FadeTransition	
`-duration: ObjectProperty<Duration>`	The duration of this transition.
`-node: ObjectProperty<Node>`	The target node of this transition.
`-fromValue: DoubleProperty`	The start opacity for this animation.
`-toValue: DoubleProperty`	The stop opacity for this animation.
`-byValue: DoubleProperty`	The incremental value on the opacity for this animation.
`+FadeTransition()`	Creates an empty `FadeTransition`.
`+FadeTransition(duration: Duration)`	Creates a `FadeTransition` with the specified duration.
`+FadeTransition(duration: Duration, node: Node)`	Creates a `FadeTransition` with the specified duration and node.

FIGURE 15.19 The **`FadeTransition`** class defines an animation for the change of opacity in a node.

Listing 15.15 gives an example that applies a fade transition to the filled color in an ellipse, as shown in Figure 15.20.

LISTING 15.15 FadeTransitionDemo.java

```
import javafx.animation.FadeTransition;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
import javafx.stage.Stage;
```

```
 9  import javafx.util.Duration;
10
11  public class FadeTransitionDemo extends Application {
12    @Override // Override the start method in the Application class
13    public void start(Stage primaryStage) {
14      // Place an ellipse to the pane
15      Pane pane = new Pane();
16      Ellipse ellipse = new Ellipse(10, 10, 100, 50);
17      ellipse.setFill(Color.RED);
18      ellipse.setStroke(Color.BLACK);
19      ellipse.centerXProperty().bind(pane.widthProperty().divide(2));
20      ellipse.centerYProperty().bind(pane.heightProperty().divide(2));
21      ellipse.radiusXProperty().bind(
22        pane.widthProperty().multiply(0.4));
23      ellipse.radiusYProperty().bind(
24        pane.heightProperty().multiply(0.4));
25      pane.getChildren().add(ellipse);
26
27      // Apply a fade transition to ellipse
28      FadeTransition ft =
29        new FadeTransition(Duration.millis(3000), ellipse);
30      ft.setFromValue(1.0);
31      ft.setToValue(0.1);
32      ft.setCycleCount(Timeline.INDEFINITE);
33      ft.setAutoReverse(true);
34      ft.play(); // Start animation
35
36      // Control animation
37      ellipse.setOnMousePressed(e -> ft.pause());
38      ellipse.setOnMouseReleased(e -> ft.play());
39
40      // Create a scene and place it in the stage
41      Scene scene = new Scene(pane, 200, 150);
42      primaryStage.setTitle("FadeTransitionDemo"); // Set the stage title
43      primaryStage.setScene(scene); // Place the scene in the stage
44      primaryStage.show(); // Display the stage
45    }
46  }
```

create a pane
create an ellipse
set ellipse fill color
set ellipse stroke color
bind ellipse properties
add ellipse to pane
create a `FadeTransition`
set start opaque value
set end opaque value
set cycle count
set auto reverse true
play animation
pause animation
resume animation

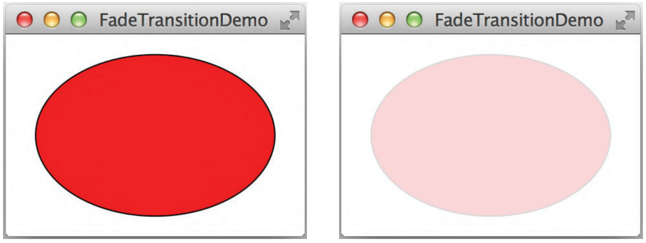


FIGURE 15.20 The `FadeTransition` animates the change of opacity in the ellipse.

The program creates a pane (line 15) and an ellipse (line 16) and places the ellipse into the pane (line 25). The ellipse's `centerX`, `centerY`, `radiusX`, and `radiusY` properties are bound to the pane's size (lines 19–24).

A fade transition is created with a duration of 3 seconds for the ellipse (line 29). It sets the start opaque to 1.0 (line 30) and the stop opaque to 0.1 (line 31). The cycle count is set to infinite so the animation is repeated indefinitely (line 32). When the mouse is pressed, the animation is paused (line 37). When the mouse is released, the animation resumes from where it was paused (line 38).

15.11.3 Timeline

`PathTransition` and **`FadeTransition`** define specialized animations. The **`Timeline`** class can be used to program any animation using one or more **`KeyFrame`**s. Each **`KeyFrame`** is executed sequentially at a specified time interval. **`Timeline`** inherits from **`Animation`**. You can construct a **`Timeline`** using the constructor **`new Timeline(KeyFrame...keyframes)`**. A **`KeyFrame`** can be constructed using

```
new KeyFrame(Duration duration, EventHandler<ActionEvent> onFinished)
```

The handler **`onFinished`** is called when the duration for the key frame is elapsed.

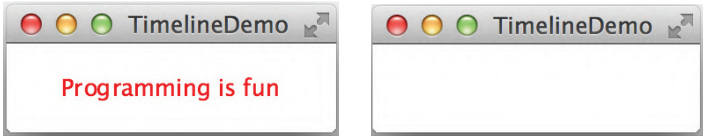
Listing 15.15 gives an example that displays a flashing text, as shown in Figure 15.21. The text is on and off alternating to animate flashing.

VideoNote
Flashing text

LISTING 15.16 TimelineDemo.java

create a stack pane
create a text
add text to pane
handler for changing text
set text empty
set text
create a `Timeline`
create a KeyFrame for handler
set cycle count indefinite
play animation
resume animation
pause animation

```
import javafx.animation.Animation;
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.util.Duration;

public class TimelineDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    StackPane pane = new StackPane();
    Text text = new Text(20, 50, "Programming is fun");
    text.setFill(Color.RED);
    pane.getChildren().add(text); // Place text into the stack pane

    // Create a handler for changing text
    EventHandler<ActionEvent> eventHandler = e -> {
      if (text.getText().length() != 0) {
        text.setText("");
      }
      else {
        text.setText("Programming is fun");
      }
    };

    // Create an animation for alternating text
    Timeline animation = new Timeline(
      new KeyFrame(Duration.millis(500), eventHandler));
    animation.setCycleCount(Timeline.INDEFINITE);
    animation.play(); // Start animation

    // Pause and resume animation
    text.setOnMouseClicked(e -> {
      if (animation.getStatus() == Animation.Status.PAUSED) {
        animation.play();
      }
      else {
        animation.pause();
45        }
46      });
47
48      // Create a scene and place it in the stage
49      Scene scene = new Scene(pane, 250, 250);
50      primaryStage.setTitle("TimelineDemo"); // Set the stage title
51      primaryStage.setScene(scene); // Place the stage in the stage
52      primaryStage.show(); // Display the stage
53    }
54  }
```



Programming is fun

FIGURE 15.21 The handler is called to set the text to "Programming is fun" or empty in turn.

The program creates a stack pane (line 17) and a text (line 18) and places the text into the pane (line 20). A handler is created to change the text to empty (lines 24–26) if it is not empty or to `Programming is fun` if it is empty (lines 27–29). A `KeyFrame` is created to run an action event in every half second (line 34). A `Timeline` animation is created to contain a key frame (lines 33 and 34). The animation is set to run indefinitely (line 35).

The mouse-clicked event is set for the text (lines 39–46). A mouse click on the text resumes the animation if the animation is paused (lines 40–42), and a mouse click on the text pauses the animation if the animation is running (lines 43–45).

In Section 14.12, Case Study: The `ClockPane` Class, you drew a clock to show the current time. The clock does not tick after it is displayed. What can you do to make the clock display a new current time every second? The key to making the clock tick is to repaint it every second with a new current time. You can use a `Timeline` to control the repainting of the clock with the code in Listing 15.17. The sample run of the program is shown in Figure 15.22.

LISTING 15.17 ClockAnimation.java

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.util.Duration;

public class ClockAnimation extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    ClockPane clock = new ClockPane(); // Create a clock

    // Create a handler for animation
    EventHandler<ActionEvent> eventHandler = e -> {
      clock.setCurrentTime(); // Set a new clock time
    };

    // Create an animation for a running clock
    Timeline animation = new Timeline(
```

create a clock

create a handler

create a time line

create a key frame
set cycle count indefinite
play animation

```
22        new KeyFrame(Duration.millis(1000), eventHandler));
23      animation.setCycleCount(Timeline.INDEFINITE);
24      animation.play(); // Start animation
25
26      // Create a scene and place it in the stage
27      Scene scene = new Scene(clock, 250, 50);
28      primaryStage.setTitle("ClockAnimation"); // Set the stage title
29      primaryStage.setScene(scene); // Place the scene in the stage
30      primaryStage.show(); // Display the stage
31    }
32  }
```

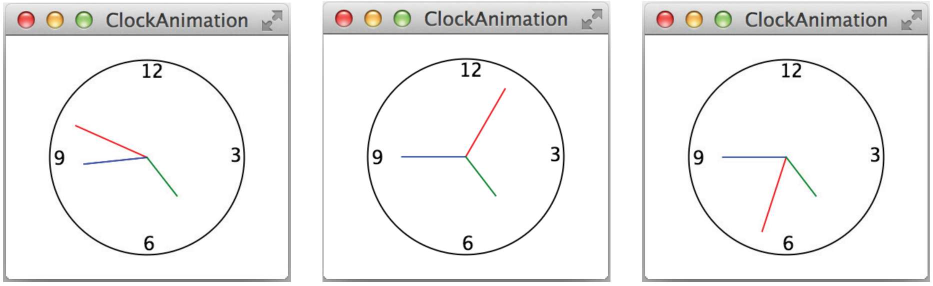


FIGURE 15.22 A live clock is displayed in the window.

The program creates an instance `clock` of `ClockPane` for displaying a clock (line 13). The `ClockPane` class is defined in Listing 14.21. The clock is placed in the scene in line 27. An event handler is created for setting the current time in the clock (lines 16–18). This handler is called every second in the key frame in the time line animation (lines 21–24). Thus, the clock time is updated every second in the animation.

15.11.1 How do you set the cycle count of an animation to infinite? How do you auto reverse an animation? How do you start, pause, and stop an animation?

15.11.2 Are `PathTransition`, `FadeTransition`, and `Timeline` subtypes of `Animation`?

15.11.3 How do you create a `PathTransition`? How do you create a `FadeTransition`? How do you create a `Timeline`?

15.11.4 How do you create a `KeyFrame`?

15.12 Case Study: Bouncing Ball

This section presents an animation that displays a ball bouncing in a pane.

The program uses `Timeline` to animate ball bouncing, as shown in Figure 15.23.

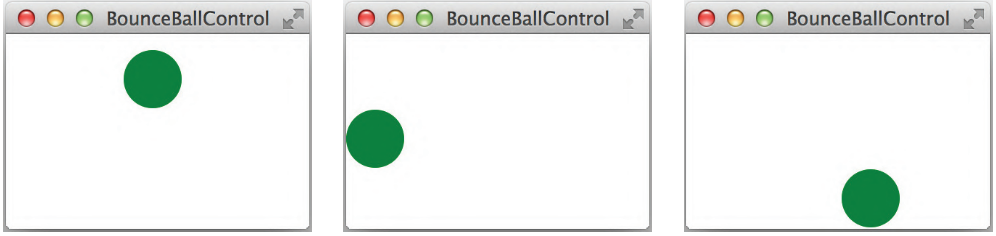


FIGURE 15.23 A ball is bouncing in a pane.

Here are the major steps to write this program:

1. Define a subclass of **`Pane`** named **`BallPane`** to display a ball bouncing, as shown in Listing 15.18.
2. Define a subclass of **`Application`** named **`BounceBallControl`** to control the bouncing ball with mouse actions, as shown in Listing 15.19. The animation pauses when the mouse is pressed, and resumes when the mouse is released. Pressing the up and down arrow keys increases/decreases the animation speed.

The relationship among these classes is shown in Figure 15.24.

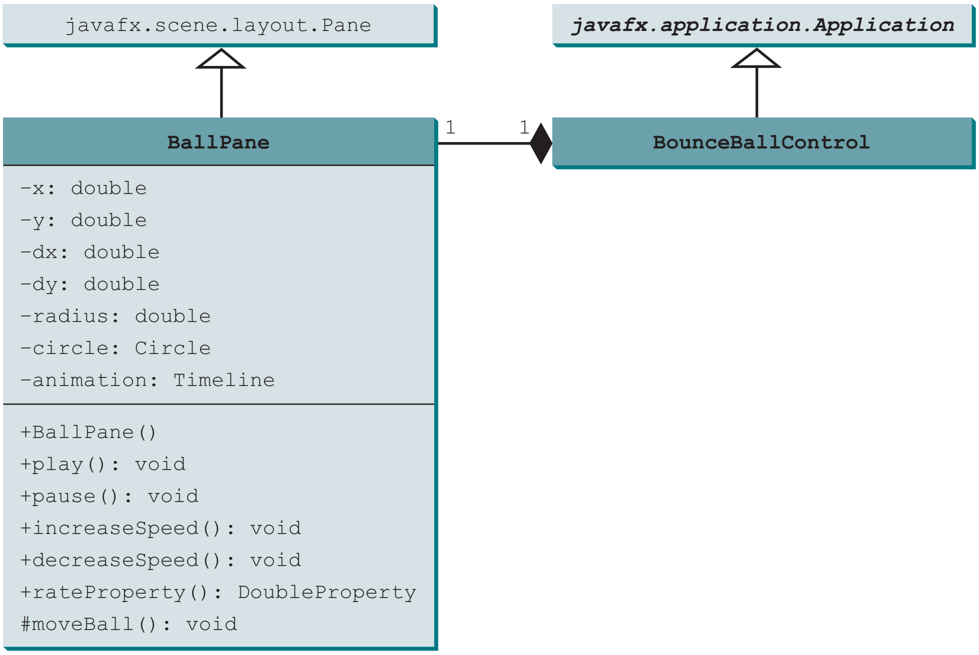

javafx.scene.layout.Pane
javafx.application.Application
BallPane
1
1
BounceBallControl
-x: double
-y: double
-dx: double
-dy: double
-radius: double
-circle: Circle
-animation: Timeline
+BallPane()
+play(): void
+pause(): void
+increaseSpeed(): void
+decreaseSpeed(): void
+rateProperty(): DoubleProperty
#moveBall(): void

FIGURE 15.24 **`BounceBallControl`** contains **`BallPane`**.

LISTING 15.18 BallPane.java

```
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.beans.property.DoubleProperty;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.util.Duration;

public class BallPane extends Pane {
  public final double radius = 20;
  private double x = radius, y = radius;
  private double dx = 1, dy = 1;
  private Circle circle = new Circle(x, y, radius);
  private Timeline animation;

  public BallPane() {
    circle.setFill(Color.GREEN); // Set ball color
    getChildren().add(circle); // Place a ball into this pane
```

create animation
keep animation running
start animation
play animation
pause animation
increase animation rate
decrease animation rate
change horizontal direction
change vertical direction
set new ball position

```
19
20     // Create an animation for moving the ball
21     animation = new Timeline(
22       new KeyFrame(Duration.millis(50), e -> moveBall()));
23     animation.setCycleCount(Timeline.INDEFINITE);
24     animation.play(); // Start animation
25   }
26
27   public void play() {
28     animation.play();
29   }
30
31   public void pause() {
32     animation.pause();
33   }
34
35   public void increaseSpeed() {
36     animation.setRate(animation.getRate() + 0.1);
37   }
38
39   public void decreaseSpeed() {
40     animation.setRate(
41       animation.getRate() > 0 ? animation.getRate() - 0.1 : 0);
42   }
43
44   public DoubleProperty rateProperty() {
45     return animation.rateProperty();
46   }
47
48   protected void moveBall() {
49     // Check boundaries
50     if (x < radius || x > getWidth() - radius) {
51       dx *= -1; // Change ball move direction
52     }
53     if (y < radius || y > getHeight() - radius) {
54       dy *= -1; // Change ball move direction
55     }
56
57     // Adjust ball position
58     x += dx;
59     y += dy;
60     circle.setCenterX(x);
61     circle.setCenterY(y);
62   }
63 }
```

`BallPane` extends `Pane` to display a moving ball (line 9). An instance of `Timeline` is created to control animation (lines 21 and 22). This instance contains a `KeyFrame` object that invokes the `moveBall()` method at a fixed rate. The `moveBall()` method moves the ball to simulate animation. The center of the ball is at (`x`, `y`), which changes to (`x + dx`, `y + dy`) on the next move (lines 58–61). When the ball is out of the horizontal boundary, the sign of `dx` is changed (from positive to negative or vice versa) (lines 50–52). This causes the ball to change its horizontal movement direction. When the ball is out of the vertical boundary, the sign of `dy` is changed (from positive to negative or vice versa) (lines 53–55). This causes the ball to change its vertical movement direction. The `pause` and `play` methods (lines 27–33) can be used to pause and resume the animation. The `increaseSpeed()` and `decreaseSpeed()` methods (lines 35–42) can be used to increase and decrease animation speed. The `rateProperty()`

method (lines 44–46) returns a binding property value for rate. This binding property will be useful for binding the rate in future applications in the next chapter.

Listing 15.19 `BounceBallControl.java`

```
 1  import javafx.application.Application;
 2  import javafx.stage.Stage;
 3  import javafx.scene.Scene;
 4  import javafx.scene.input.KeyCode;
 5
 6  public class BounceBallControl extends Application {
 7    @Override // Override the start method in the Application class
 8    public void start(Stage primaryStage) {
 9      BallPane ballPane = new BallPane(); // Create a ball pane
10
11      // Pause and resume animation
12      ballPane.setOnMousePressed(e -> ballPane.pause());
13      ballPane.setOnMouseReleased(e -> ballPane.play());
14
15      // Increase and decrease animation
16      ballPane.setOnKeyPressed(e -> {
17        if (e.getCode() == KeyCode.UP) {
18          ballPane.increaseSpeed();
19        }
20        else if (e.getCode() == KeyCode.DOWN) {
21          ballPane.decreaseSpeed();
22        }
23      });
24
25      // Create a scene and place it in the stage
26      Scene scene = new Scene(ballPane, 250, 150);
27      primaryStage.setTitle("BounceBallControl"); // Set the stage title
28      primaryStage.setScene(scene); // Place the scene in the stage
29      primaryStage.show(); // Display the stage
30
31      // Must request focus after the primary stage is displayed
32      ballPane.requestFocus();
33    }
34  }
```

create a ball pane
pause animation
resume animation
increase speed
decrease speed
request focus on pane

The **`BounceBallControl`** class is the main JavaFX class that extends **`Application`** to display the ball pane with control functions. The mouse-pressed and mouse-released handlers are implemented for the ball pane to pause the animation and resume the animation (lines 12 and 13). When the UP arrow key is pressed, the ball pane's **`increaseSpeed()`** method is invoked to increase the ball's movement (line 18). When the down arrow key is pressed, the ball pane's **`decreaseSpeed()`** method is invoked to reduce the ball's movement (line 21).

Invoking **`ballPane.requestFocus()`** in line 32 sets the input focus to **`ballPane`**.

Check Point

15.12.1 How does the program make the ball appear to be moving?

15.12.2 How does the code in Listing 15.17, BallPane.java, change the direction of the ball movement?

15.12.3 What does the program do when the mouse is pressed on the ball pane? What does the program do when the mouse is released on the ball pane?

15.12.4 If line 32 in Listing 15.18, BounceBall.java, is not in the program, what would happen when you press the up or the down arrow key?

15.12.5 If line 23 is not in Listing 15.17, what would happen?

15.13 Case Study: US Map

Key Point

This section presents a program that draws, colors, and resizes a US map.

The program reads the GPS coordinates for each state in the 48 continental United States, and draws a polygon to connect the coordinates and displays all the polygons, as shown in Figure 15.25.

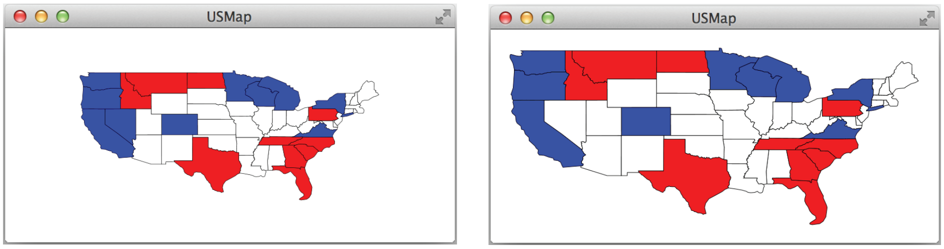


FIGURE 15.25 The program displays, colors, and resizes the US map.

The coordinates are contained in a file at https://liveexample.pearsoncmg.com/data/usmap.txt. For each state, the file contains the state name (e.g., Alabama) and all the coordinates (latitude and longitude) for the state. For example, the following is an example for Alabama and Arkansas:

```
Alabama
  35.0041 -88.1955
  34.9918 -85.6068
  ...
  34.9479 -88.1721
  34.9107 -88.1461
Arkansas
  33.0225 -94.0416
  33.0075 -91.2057
  ...
```

A polygon is displayed in red, blue, or white when the primary, secondary, or middle mouse button is clicked in the polygon. The map size is increased when the up arrow key is pressed, and decreased when the down arrow key is pressed. Listing 15.20 gives the code for this program.

LISTING 15.20 USMap.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Polygon;
import javafx.scene.Group;
import javafx.scene.layout.BorderPane;
import javafx.scene.input.*;
import javafx.geometry.Point2D;
import java.util.*;

public class USMap extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    MapPane map = new MapPane();
```

create a map

```
    Scene scene = new Scene(map, 1200, 800);
    primaryStage.setTitle("USMap"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    map.setOnKeyPressed(e -> {
      if (e.getCode() == KeyCode.UP) {
        map.enlarge(); // Enlarge the map
      }
      else if (e.getCode() == KeyCode.DOWN) {
        map.shrink(); // SHrink the map
      }
    });
    map.requestFocus();
  }

  class MapPane extends BorderPane {
    private Group group = new Group();

    MapPane() {
      // Load coordinates from a file
      ArrayList<ArrayList<Point2D>> points = getPoints();

      // Add points to the polygon list
      for (int i = 0; i < points.size(); i++) {
        Polygon polygon = new Polygon();
        // Add points to the polygon list
        for (int j = 0; j < points.get(i).size(); j++)
          polygon.getPoints().addAll(points.get(i).get(j).getX(),
            -points.get(i).get(j).getY());
        polygon.setFill(Color.WHITE);
        polygon.setStroke(Color.BLACK);
        polygon.setStrokeWidth(1 / 14.0);

        polygon.setOnMouseClicked(e -> {
          if (e.getButton() == MouseButton.PRIMARY) {
            polygon.setFill(Color.RED);
          }
          else if (e.getButton() == MouseButton.SECONDARY) {
            polygon.setFill(Color.BLUE);
          }
          else {
            polygon.setFill(Color.WHITE);
          }
        });

        group.getChildren().add(polygon);
      }

      group.setScaleX(14);
      group.setScaleY(14);
      this.setCenter(group);
    }

    public void enlarge() {
      group.setScaleX(1.1 * group.getScaleX());
      group.setScaleY(1.1 * group.getScaleY());
    }

    public void shrink() {
```

listen to key event

enlarge map

shrink map

request focus

extends `BorderPane`

create a `Group`

get coordinates for state

add coordinates

set polygon stroke width

set listener for mouse click

color polygon

add a polygon to group

scale polygon

center group in the map

enlarge map

shrink map

create array list

try-with-resource

open an Internet resource

read a string

start a state

create a state list

read latitude value

read longitude value

add a point to list

return list of points

```
 76    group.setScaleX(0.9 * group.getScaleX());
 77    group.setScaleY(0.9 * group.getScaleY());
 78   }
 79
 80   private ArrayList<ArrayList<Point2D>> getPoints() {
 81    ArrayList<ArrayList<Point2D>> points = new ArrayList<>();
 82
 83    try (Scanner input = new Scanner(new java.net.URL(
 84     "https://liveexample.pearsoncmg.com/data/usmap.txt")
 85      .openStream())) {
 86     while (input.hasNext()) {
 87      String s = input.nextLine();
 88      if (Character.isAlphabetic(s.charAt(0))) {
 89       points.add(new ArrayList<>()); // For a new state
 90      }
 91      else {
 92       Scanner scanAString = new Scanner(s); // Scan one point
 93       double y = scanAString.nextDouble();
 94       double x = scanAString.nextDouble();
 95       points.get(points.size() - 1).add(new Point2D(x, y));
 96      }
 97     }
 98    }
 99    catch (Exception ex) {
100     ex.printStackTrace();
101    }
102
103    return points;
104   }
105  }
106 }
```

the `Group` class

the `scaleX` property

the `scaleY` property

The program defines **`MapPane`** that extends **`BorderPane`** to display a map in the center of the border pane (line 32). The program needs to resize the polygons in the map. An instance of the **`Group`** class is created to hold all the polygons (line 33). Grouping the polygons enables all polygons to be resized in one operation. Resizing the group will cause all polygons in the group to resize accordingly. Resizing can be done by applying the **`scaleX`** and **`scaleY`** properties in the group (lines 65 and 66).

The **`getPoints()`** method is used to return all the coordinates in an array list (line 80). The array list consists of sublists. Each sublist contains the coordinates for a state and is added to the array list (line 89). A **`Point2D`** object represents the *x*- and *y*-coordinates of the point (line 81). The method creates a **`Scanner`** object to read data for the map coordinates from a file on the Internet (lines 83–85). The program reads lines from the file. For each line, if the first character is an alphabet, the line is for a new state name (line 88) and a new sublist is created and added to the **`points`** array list (line 89). Otherwise, the line contains the two coordinates. The latitude becomes the *y*-coordinate for the point (line 93), and the longitude corresponds to the *x*-coordinate of the point (line 94). The program stores the points for a state in a sublist (line 95). **`points`** is an array list that contains **`48`** sublists.

the `scaleX` property

the `scaleY` property

The constructor of **`MapPane`** obtains sublists of the coordinates from the file (line 37). For each sublist of the points, a polygon is created (line 41). The points are added to the polygon (lines 43–45). Since the *y*-coordinates increase upward in the conventional coordinate system, but downward in the Java coordinate system, the program changes the sign for the *y*-coordinates in line 45. The polygon properties are set in lines 46–48. Note the **`strokeWidth`** is set to **`1 / 14.0`** (line 48) because all the polygons are scaled up **`14`** times in lines 65 and 66. If the **`strokeWidth`** is not set to this value, the stroke width will be very thick. Since polygons are very small, applying the **`setScaleX`** and **`setScaleY`** methods on the group causes all the nodes inside the group to be enlarged (lines 65 and 66). `MapPane` is a `BorderPane`. The group is placed in the center of the border pane (line 67).

The `enlarge()` and `shrink()` methods are defined in `MapPane` (lines 70–78). They can be called to enlarge or shrink the group to cause all the polygons in the group to scale up or down.

Each polygon is set to listen to mouse-clicked event (lines 50–60). When clicking the primary/secondary/middle mouse button on a polygon, the polygon is filled red/blue/white.

The program creates an instance of `MapPane` (line 15) and places it in the scene (line 16). The map listens to the key-pressed event to enlarge or shrink the map upon pressing the up and down arrow key (lines 21–28). Since the map is inside the scene, invoking `map.requestFocus()` enables the map to receive key events (line 29).

Check Point

15.13.1 What would happen if line 29 in Listing 15.20 is removed?

15.13.2 What would happen if `map` is replaced by `scene` in line 21 in Listing 15.20?

15.13.3 What would happen if `map` is replaced by `primaryStage` in line 21 in Listing 15.20?

KEY TERMS

anonymous inner class	602
event	596
event-driven programming	596
event handler	597
event–handler interface	597
event object	596
event source object	596
functional interface	607
inner class	599
key code	613
lambda expression	605
observable object	616
single abstract method interface	607

CHAPTER SUMMARY

1. The root class of the JavaFX event classes is `javafx.event.Event`, which is a subclass of `java.util.EventObject`. The subclasses of `Event` deal with special types of events, such as action events, window events, mouse events, and key events. If a node can fire an event, any subclass of the node can fire the same type of event.

2. The handler object's class must implement the corresponding *event–handler interface*. JavaFX provides a handler interface `EventHandler<T extends Event>` for every event class `T`. The handler interface contains the `handle(T e)` method for handling event `e`.

3. The handler object must be registered by the *source object*. Registration methods depend on the event type. For an action event, the method is `setOnAction`. For a mouse-pressed event, the method is `setOnMousePressed`. For a key-pressed event, the method is `setOnKeyPressed`.

4. An *inner class*, or *nested class*, is defined within the scope of another class. An inner class can reference the data and methods defined in the outer class in which it nests, so you need not pass the reference of the outer class to the constructor of the inner class.

5. An anonymous inner class can be used to shorten the code for event handling. Furthermore, a lambda expression can be used to greatly simplify the event-handling code for functional interface handlers.

6. A *functional interface* is an interface with exactly one abstract method. This is also known as a single abstract method (SAM) interface.

7. A `MouseEvent` is fired whenever a mouse button is pressed, released, clicked, moved, or dragged on a node or a scene. The `getButton()` method can be used to detect which mouse button is pressed for the event.

8. A `KeyEvent` is fired whenever a key is pressed, released, or typed on a node or a scene. The `getCode()` method can be used to return the code value for the key.

9. An instance of `Observable` is known as an observable object, which contains the `add-Listener(InvalidationListener listener)` method for adding a listener. Once the value is changed in the property, a listener is notified. The listener class should implement the `InvalidationListener` interface, which uses the `invalidated` method to handle the property value change.

10. The abstract `Animation` class provides the core functionalities for animations in JavaFX. `PathTransition`, `FadeTransition`, and `Timeline` are specialized classes for implementing animations.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

Sections 15.2–15.7

***15.1** (*Pick four cards*) Write a program that lets the user click the *Refresh* button to display four cards from a deck of 52 cards, as shown in Figure 15.26a. (See the hint in Programming Exercise 14.3 on how to obtain four random cards.)

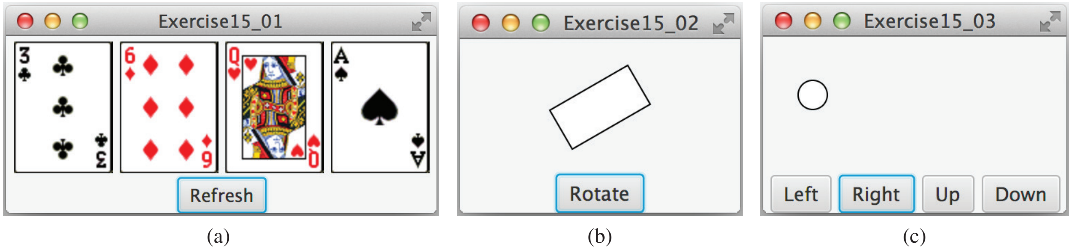


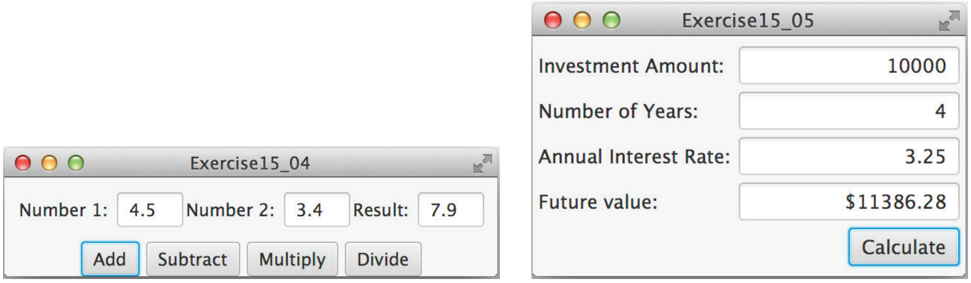
FIGURE 15.26 (a) Exercise 15.1 displays four cards randomly. *Source*: Fotolia. (b) Exercise 15.2 rotates the rectangle. (c) Exercise 15.3 uses the buttons to move the ball.

15.2 (*Rotate a rectangle*) Write a program that rotates a rectangle 15 degrees to the right when the *Rotate* button is clicked, as shown in Figure 15.26b.

***15.3** (*Move the ball*) Write a program that moves the ball in a pane. You should define a pane class for displaying the ball and provide the methods for moving the ball left, right, up, and down, as shown in Figure 15.26c. Check the boundary to prevent the ball from moving out of sight completely.

VideoNote
Simple calculator

***15.4** (*Create a simple calculator*) Write a program to perform addition, subtraction, multiplication, and division, as shown in Figure 15.27a.



Number 1: 4.5 Number 2: 3.4 Result: 7.9

Investment Amount: 10000
Number of Years: 4
Annual Interest Rate: 3.25
Future value: $11386.28

(a) (b)

FIGURE 15.27 (a) Exercise 15.4 performs addition, subtraction, multiplication, and division on double numbers. (b) The user enters the investment amount, years, and interest rate to compute future value.

***15.5** (*Create an investment-value calculator*) Write a program that calculates the future value of an investment at a given interest rate for a specified number of years. The formula for the calculation is

$$\texttt{futureValue = investmentAmount * (1 + monthlyInterestRate)}^{\texttt{years*12}}$$

Use text fields for the investment amount, number of years, and annual interest rate. Display the future amount in a text field when the user clicks the *Calculate* button, as shown in Figure 15.27b.

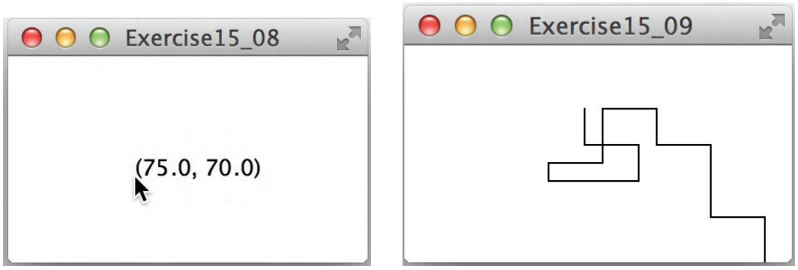
Sections 15.8 and 15.9

****15.6** (*Alternate two messages*) Write a program to display the text `Java is fun` and `Java is powerful` alternately with a mouse click.

***15.7** (*Change color using a mouse*) Write a program that displays the color of a circle as black when the mouse button is pressed, and as white when the mouse button is released.

***15.8** (*Display the mouse position*) Write two programs, such that one displays the mouse position when the mouse button is clicked (see Figure 15.28a), and the other displays the mouse position when the mouse button is pressed and ceases to display it when the mouse button is released.

***15.9** (*Draw lines using the arrow keys*) Write a program that draws line segments using the arrow keys. The line starts from (100, 100) in the pane and draws toward east, north, west, or south when the right-arrow key, up-arrow key, left-arrow key, or down-arrow key is pressed, as shown in Figure 15.28b.



(75.0, 70.0)

(a) (b)

FIGURE 15.28 (a) Exercise 15.8 displays the mouse position. (b) Exercise 15.9 uses the arrow keys to draw the lines.

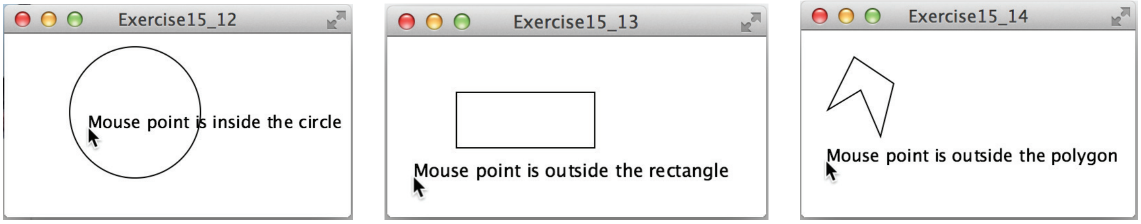
****15.10** (*Enter and display a string*) Write a program that receives a string from the keyboard and displays it on a pane. The *Enter* key signals the end of a string. Whenever a new string is entered, it is displayed on the pane.

***15.11** (*Move a circle using keys*) Write a program that moves a circle up, down, left, or right using the arrow keys.

VideoNote
Check mouse-point location

****15.12** (*Geometry: inside a circle?*) Write a program that draws a fixed circle centered at (`100`, `60`) with radius `50`. Whenever the mouse is moved, display a message indicating whether the mouse point is inside the circle at the mouse point or outside of it, as shown in Figure 15.29a.

****15.13** (*Geometry: inside a rectangle?*) Write a program that draws a fixed rectangle centered at (`100`, `60`) with width `100` and height `40`. Whenever the mouse is moved, display a message indicating whether the mouse point is inside the rectangle at the mouse point or outside of it, as shown in Figure 15.29b. To detect whether a point is inside a polygon, use the `contains` method defined in the `Node` class.



Mouse point is inside the circle

Mouse point is outside the rectangle

Mouse point is outside the polygon

(a) (b) (c)

FIGURE 15.29 Detect whether a point is inside a circle, a rectangle, or a polygon.

****15.14** (*Geometry: inside a polygon?*) Write a program that draws a fixed polygon with points at (`40`, `20`), (`70`, `40`), (`60`, `80`), (`45`, `45`), and (`20`, `60`). Whenever the mouse is moved, display a message indicating whether the mouse point is inside the polygon at the mouse point or outside of it, as shown in Figure 15.29c. To detect whether a point is inside a polygon, use the `contains` method defined in the `Node` class.

****15.15** (*Geometry: add and remove points*) Write a program that lets the user click on a pane to dynamically create and remove points (see Figure 15.30a). When the user left-clicks the mouse (primary button), a point is created and displayed at the mouse point. The user can remove a point by pointing to it and right-clicking the mouse (secondary button).

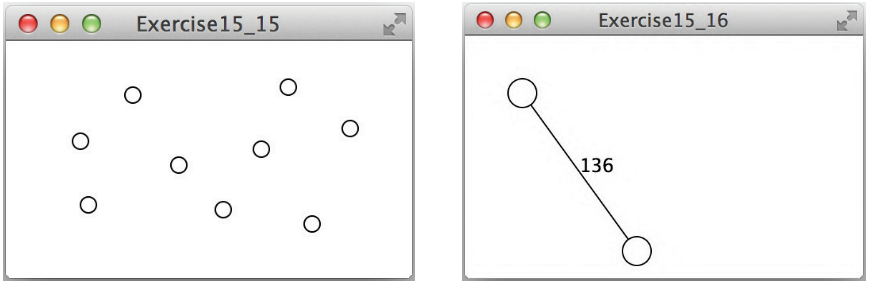
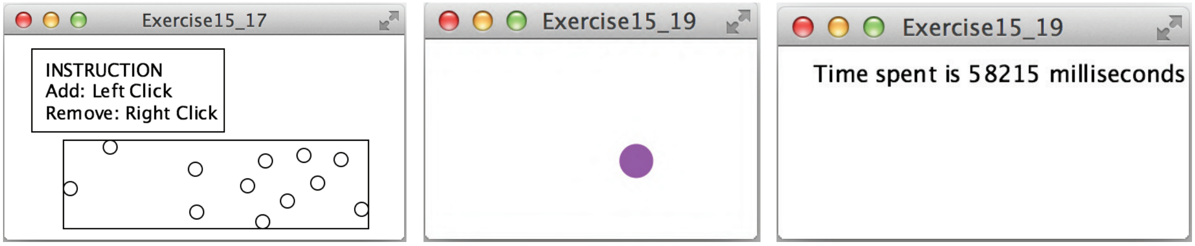


FIGURE 15.30 (a) Exercise 15.15 allows the user to create/remove points dynamically. (b) Exercise 15.16 displays two vertices and a connecting edge.

***15.16** (*Two movable vertices and their distances*) Write a program that displays two circles with radius **10** at location (**40**, **40**) and (**120**, **150**) with a line connecting the two circles, as shown in Figure 15.30b. The distance between the circles is displayed along the line. The user can drag a circle. When that happens, the circle and its line are moved, and the distance between the circles is updated.

****15.17** (*Geometry: find the bounding rectangle*) Write a program that enables the user to add and remove points in a two-dimensional plane dynamically, as shown in Figure 15.31a. A minimum bounding rectangle is updated as the points are added and removed. Assume the radius of each point is **10** pixels.



INSTRUCTION
Add: Left Click
Remove: Right Click

```
Time spent is 58215 milliseconds
```

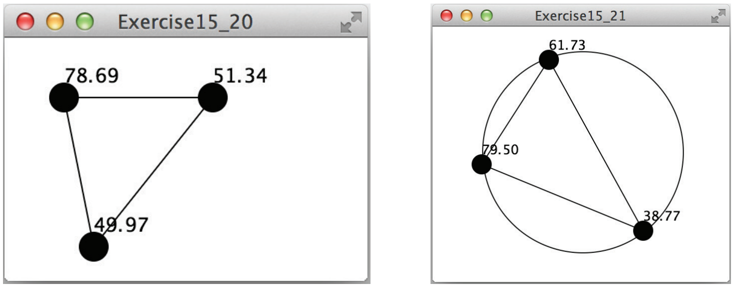
(a) (b) (c)

FIGURE 15.31 (a) Exercise 15.17 enables the user to add/remove points dynamically and displays the bounding rectangle. (b) When you click a circle, a new circle is displayed at a random location. (c) After 20 circles are clicked, the time spent is displayed in the pane.

****15.18** (*Move a rectangle using mouse*) Write a program that displays a rectangle. You can point the mouse inside the rectangle and drag (i.e., move with mouse pressed) the rectangle wherever the mouse goes. The mouse point becomes the center of the rectangle.

****15.19** (*Game: eye–hand coordination*) Write a program that displays a circle of radius **10** pixels filled with a random color at a random location on a pane, as shown in Figure 15.31b. When you click the circle, it disappears and a new random-color circle is displayed at another random location. After 20 circles are clicked, display the time spent in the pane, as shown in Figure 15.31c.

****15.20** (*Geometry: display angles*) Write a program that enables the user to drag the vertices of a triangle and displays the angles dynamically as the triangle shape changes, as shown in Figure 15.32a. The formula to compute angles is given in Listing 4.1.



(a) (b)

FIGURE 15.32 (a) Exercise 15.20 enables the user to drag vertices and display the angles dynamically. (b) Exercise 15.21 enables the user to drag vertices and display the angles in the triangle dynamically.

***15.21** (*Drag points*) Draw a circle with three random points on the circle. Connect the points to form a triangle. Display the angles in the triangle. Use the mouse to drag a point along the perimeter of the circle. As you drag it, the triangle and angles are redisplayed dynamically, as shown in Figure 15.32b. For computing angles in a triangle, see Listing 4.1.

Section 15.10

***15.22** (*Auto resize cylinder*) Rewrite Programming Exercise 14.10 so the cylinder's width and height are automatically resized when the window is resized.

***15.23** (*Auto resize stop sign*) Rewrite Programming Exercise 14.15 so the stop sign's width and height are automatically resized when the window is resized.

Section 15.11

****15.24** (*Animation: pendulum swing*) Write a program that animates a pendulum swing, as shown in Figure 15.33. Press/release the mouse to pause/resume the animation.

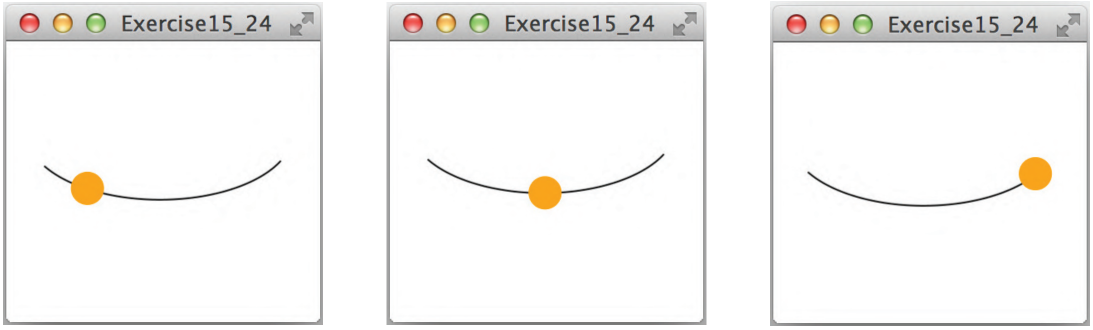


FIGURE 15.33 The program animates a pendulum swing.

****15.25** (*Animation: ball on curve*) Write a program that animates a ball moving along a sine curve, as shown in Figure 15.34. When the ball gets to the right border, it starts over from the left. Enable the user to resume/pause the animation with a click on the left/right mouse button.

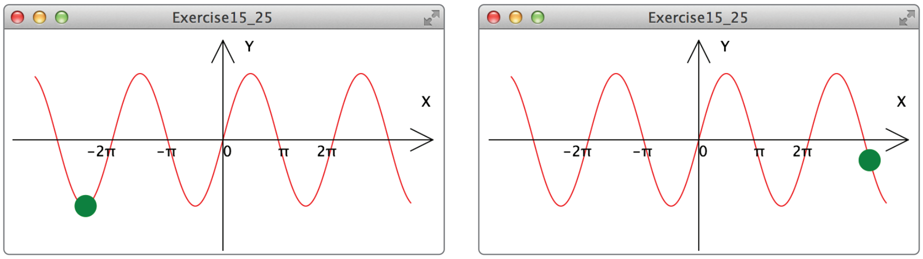
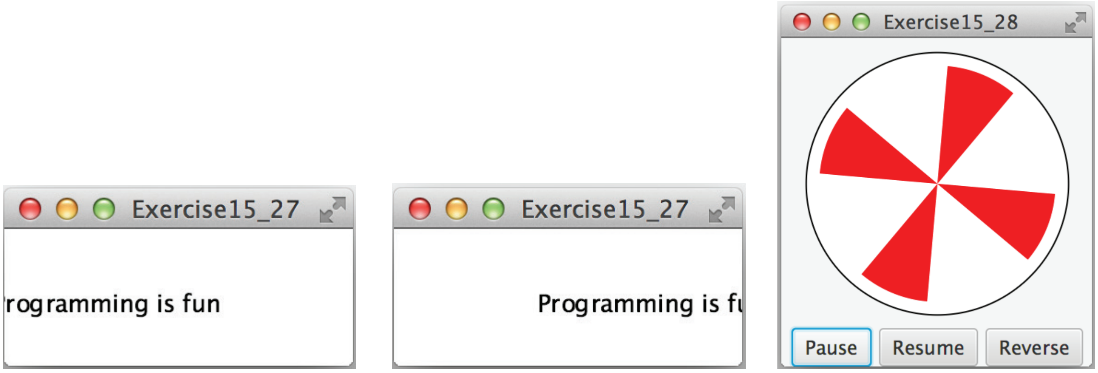


FIGURE 15.34 The program animates a ball traveling along a sine curve.

***15.26** (*Change opacity*) Rewrite Programming Exercise 15.24 so the ball's opacity is changed as it swings.

***15.27** (*Control a moving text*) Write a program that displays a moving text, as shown in Figures 15.35a and b. The text moves from left to right circularly. When it disappears in the right, it reappears from the left. The text freezes when the mouse is pressed, and moves again when the button is released.



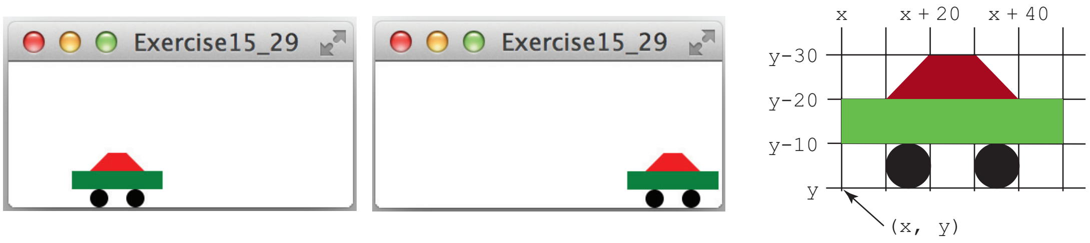
(a) (b) (c)

FIGURE 15.35 (a and b) A text is moving from left to right circularly. (c) The program simulates a fan running.

****15.28** (*Display a running fan*) Write a program that displays a running fan, as shown in Figure 15.35c. Use the *Pause*, *Resume*, and *Reverse* buttons to pause, resume, and reverse fan running.

VideoNote
Display a running fan

****15.29** (*Racing car*) Write a program that simulates car racing, as shown in Figure 15.36a. The car moves from left to right. When it hits the right end, it restarts from the left and continues the same process. You can use a timer to control animation. Redraw the car with new base coordinates (*x*, *y*), as shown in Figure 15.36b. Also let the user pause/resume the animation with a button press/release and increase/decrease the car speed by pressing the up and down arrow keys.



(a) (b)

FIGURE 15.36 (a) The program displays a moving car. (b) You can redraw a car with a new base point.

****15.30** (*Slide show*) Twenty-five slides are stored as image files (**slide0.jpg**, **slide1.jpg**, . . . , **slide24.jpg**) in the **image** directory downloadable along with the source code in the book. The size of each image is 800 × 600. Write a program that automatically displays the slides repeatedly. Each slide is shown for two seconds. The slides are displayed in order. When the last slide finishes, the first slide is redisplayed, and so on. Click to pause if the animation is currently playing. Click to resume if the animation is currently paused.

****15.31** (*Geometry: pendulum*) Write a program that animates a pendulum swinging, as shown in Figure 15.37. Press the up arrow key to increase the speed, and the down arrow key to decrease it. Press the *S* key to stop animation of and the *R* key to resume it.

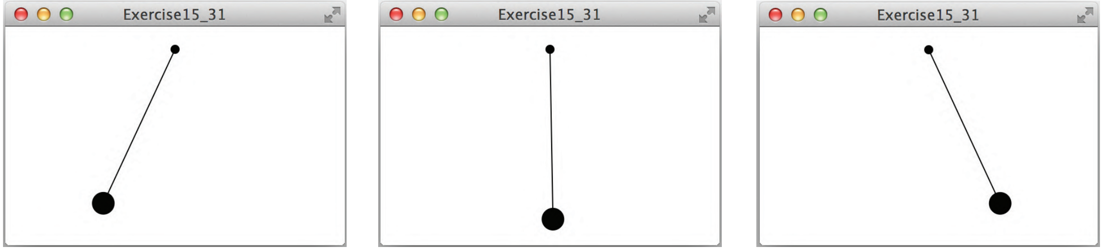

Exercise15_31
Exercise15_31
Exercise15_31

FIGURE 15.37 Exercise 15.31 animates a pendulum swinging.

***15.32** (*Control a clock*) Modify Listing 14.21, ClockPane.java, to add the animation into this class and add two methods `start()` and `stop()` to start and stop the clock, respectively. Write a program that lets the user control the clock with the *Start* and *Stop* buttons, as shown in Figure 15.38a.

*****15.33** (*Game: bean-machine animation*) Write a program that animates the bean machine introduced in Programming Exercise 7.37. The animation terminates after 10 balls are dropped, as shown in Figures 15.38b and c.

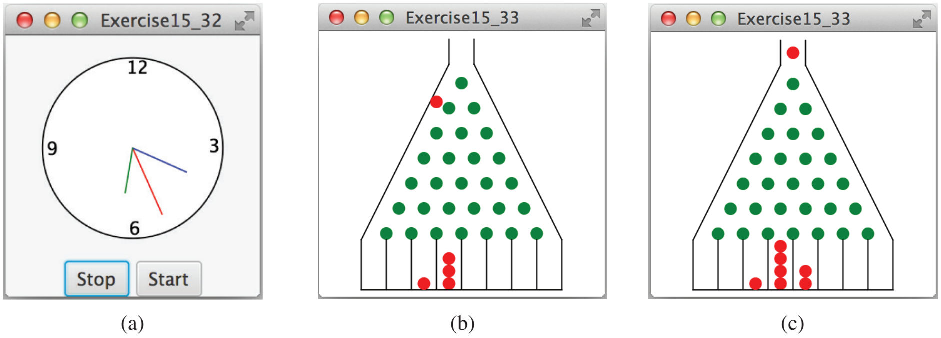

Exercise15_32
12
9
3
6
Stop
Start
Exercise15_33
Exercise15_33
(a)
(b)
(c)

FIGURE 15.38 (a) Exercise 15.32 allows the user to start and stop a clock. (b and c) The balls are dropped into the bean machine.

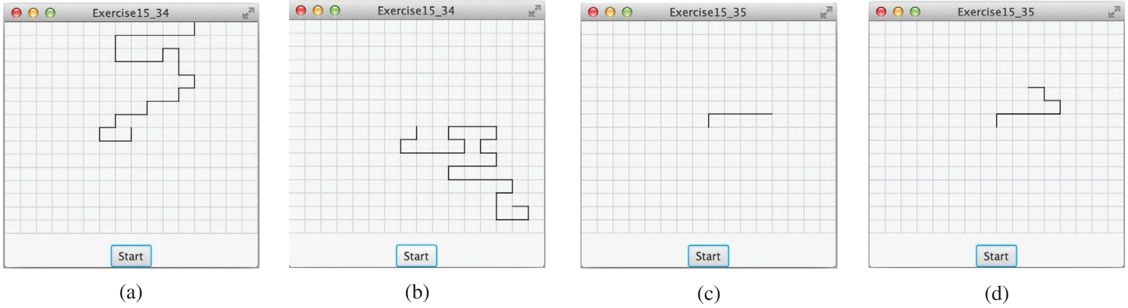

Exercise15_34
Start
Exercise15_34
Start
Exercise15_35
Start
Exercise15_35
Start
(a)
(b)
(c)
(d)

FIGURE 15.39 (a) A path ends at a boundary point. (b) A path ends at dead-end point. (c and d) Animation shows the progress of a path step by step.

*****15.34** (*Simulation: self-avoiding random walk*) A self-avoiding walk in a lattice is a path from one point to another that does not visit the same point twice. Self-avoiding walks have applications in physics, chemistry, and mathematics. They can be used to model chain-like entities such as solvents and polymers. Write a program that displays a random path that starts from the center and ends at a point on the boundary, as shown in Figure 15.39a, or ends at a dead-end point (i.e., surrounded by four points that have already been visited), as shown in Figure 15.39b. Assume the size of the lattice is `16` by `16`.

*****15.35** (*Animation: self-avoiding random walk*) Revise the preceding exercise to display the walk step by step in an animation, as shown in Figures 15.39c and d.

****15.36** (*Simulation: self-avoiding random walk*) Write a simulation program to show that the chance of getting dead-end paths increases as the grid size increases. Your program simulates lattices with size from 10 to 80 with increments of 5. For each lattice size, simulate a self-avoiding random walk 10,000 times and display the probability of the dead-end paths, as shown in the following sample output:

```
For a lattice of size 10, the probability of dead-end paths is 10.6%
For a lattice of size 15, the probability of dead-end paths is 14.0%
...
For a lattice of size 80, the probability of dead-end paths is 99.5%
```

CHAPTER

16

JAVAFX UI CONTROLS AND MULTIMEDIA

Objectives

- To create graphical user interfaces with various user-interface controls (§§16.2–16.11).
- To create a label with text and graphics using the `Label` class, and explore properties in the abstract `Labeled` class (§16.2).
- To create a button with text and graphic using the `Button` class, and set a handler using the `setOnAction` method in the abstract `ButtonBase` class (§16.3).
- To create a check box using the `CheckBox` class (§16.4).
- To create a radio button using the `RadioButton` class, and group radio buttons using a `ToggleGroup` (§16.5).
- To enter data using the `TextField` class and password using the `PasswordField` class (§16.6).
- To enter data in multiple lines using the `TextArea` class (§16.7).
- To select a single item using `ComboBox` (§16.8).
- To select a single or multiple items using `ListView` (§16.9).
- To select a range of values using `ScrollBar` (§16.10).
- To select a range of values using `Slider` and explore differences between `ScrollBar` and `Slider` (§16.11).
- To develop a tic-tac-toe game (§16.12).
- To view and play video and audio using the `Media`, `MediaPlayer`, and `MediaView` (§16.13).
- To develop a case study for showing the national flag and playing the national anthem (§16.14).

16.1 Introduction

JavaFX provides many UI controls for developing a comprehensive user interface.

GUI

A graphical user interface (GUI) makes a program user-friendly and easy to use. Creating a GUI requires creativity and knowledge of how UI controls work. Since the UI controls in JavaFX are very flexible and versatile, you can create a wide assortment of useful user interfaces for rich GUI applications.

Oracle provides tools for visually designing and developing GUIs. This enables the programmer to rapidly assemble the elements of a GUI with minimum coding. Tools, however, cannot do everything. You have to modify the programs they produce. Consequently, before you begin to use the visual tools, you must understand the basic concepts of JavaFX GUI programming.

Previous chapters used UI controls such as **`Button`**, **`Label`**, and **`TextField`**. This chapter introduces the frequently used UI controls in detail (see Figure 16.1).

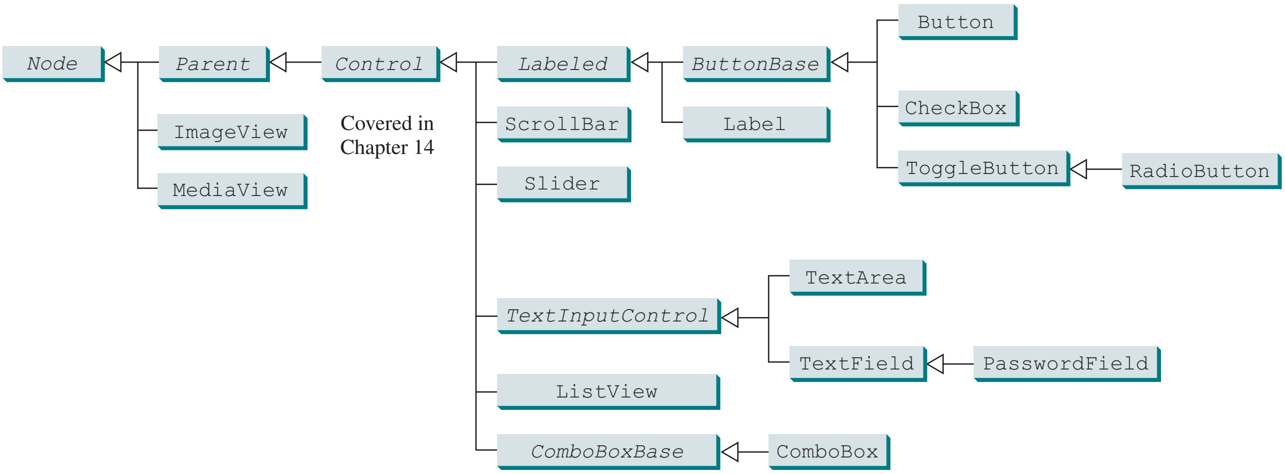

Node
Parent
Control
Labeled
ButtonBase
Button
CheckBox
ToggleButton
RadioButton
Label
ScrollBar
Slider
ImageView
MediaView
Covered in Chapter 14
TextInputControl
TextArea
TextField
PasswordField
ListView
ComboBoxBase
ComboBox

FIGURE 16.1 These UI controls are frequently used to create user interfaces.

naming convention for controls

Note

Throughout this book, the prefixes **`lbl`**, **`bt`**, **`chk`**, **`rb`**, **`tf`**, **`pf`**, **`ta`**, **`cbo`**, **`lv`**, **`scb`**, **`sld`**, and **`mp`** are used to name reference variables for **`Label`**, **`Button`**, **`CheckBox`**, **`RadioButton`**, **`TextField`**, **`PasswordField`**, **`TextArea`**, **`ComboBox`**, **`ListView`**, **`ScrollBar`**, **`Slider`**, and **`MediaPlayer`**, respectively.

16.2 Labeled and Label

A label *is a display area for a short text, a node, or both. It is often used to label other controls (usually text fields).*

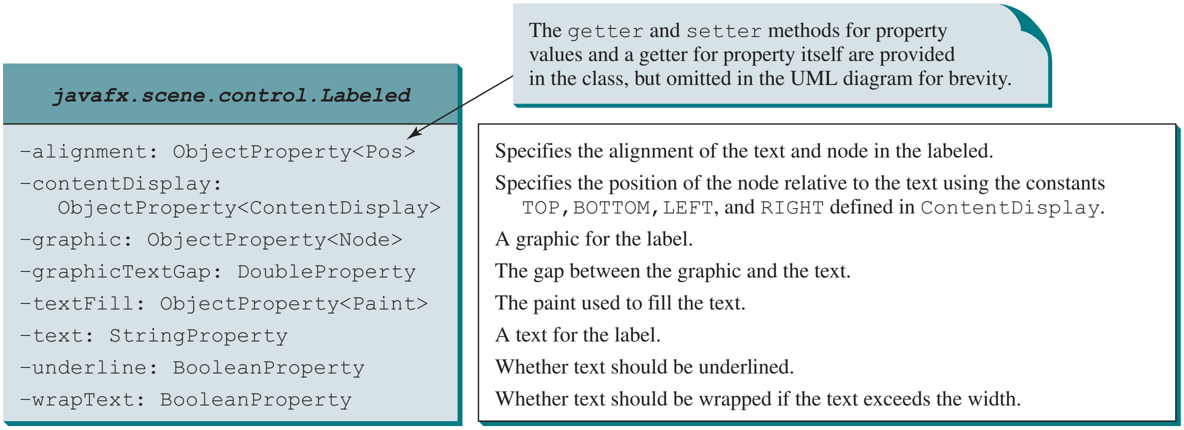
Labels and buttons share many common properties. These common properties are defined in the **`Labeled`** class, as shown in Figure 16.2.

A **`Label`** can be constructed using one of the three constructors shown in Figure 16.3.

The **`graphic`** property can be any node such as a shape, an image, or a control. Listing 16.1 gives an example that displays several labels with text and images in the label, as shown in Figure 16.4.

LISTING 16.1 LabelWithGraphic.java

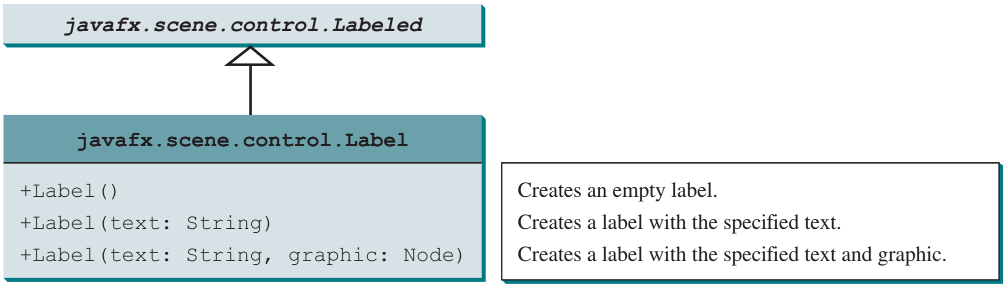
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.ContentDisplay;
```


javafx.scene.control.Labeled
-alignment: ObjectProperty<Pos>
-contentDisplay: ObjectProperty<ContentDisplay>
-graphic: ObjectProperty<Node>
-graphicTextGap: DoubleProperty
-textFill: ObjectProperty<Paint>
-text: StringProperty
-underline: BooleanProperty
-wrapText: BooleanProperty

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Specifies the alignment of the text and node in the labeled.
Specifies the position of the node relative to the text using the constants `TOP`, `BOTTOM`, `LEFT`, and `RIGHT` defined in `ContentDisplay`.
A graphic for the label.
The gap between the graphic and the text.
The paint used to fill the text.
A text for the label.
Whether text should be underlined.
Whether text should be wrapped if the text exceeds the width.

FIGURE 16.2 `Labeled` defines common properties for `Label`, `Button`, `CheckBox`, and `RadioButton`.


javafx.scene.control.Labeled
javafx.scene.control.Label
+Label()
+Label(text: String)
+Label(text: String, graphic: Node)

Creates an empty label.
Creates a label with the specified text.
Creates a label with the specified text and graphic.

FIGURE 16.3 `Label` is created to display a text or a node, or both.

```java
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
import javafx.scene.shape.Ellipse;

public class LabelWithGraphic extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    ImageView us = new ImageView(new Image("image/us.gif"));
    Label lb1 = new Label("US\n50 States", us);
    lb1.setStyle("-fx-border-color: green; -fx-border-width: 2");
    lb1.setContentDisplay(ContentDisplay.BOTTOM);
    lb1.setTextFill(Color.RED);

    Label lb2 = new Label("Circle", new Circle(50, 50, 25));
    lb2.setContentDisplay(ContentDisplay.TOP);
    lb2.setTextFill(Color.ORANGE);

    Label lb3 = new Label("Rectangle", new Rectangle(10, 10, 50, 25));
    lb3.setContentDisplay(ContentDisplay.RIGHT);

    Label lb4 = new Label("Ellipse", new Ellipse(50, 50, 50, 25));
    lb4.setContentDisplay(ContentDisplay.LEFT);

```

create a label
set node position
create a label
set node position
create a label
create a label

```
34     Ellipse ellipse = new Ellipse(50, 50, 50, 25);
35     ellipse.setStroke(Color.GREEN);
36     ellipse.setFill(Color.WHITE);
37     StackPane stackPane = new StackPane();
38     stackPane.getChildren().addAll(ellipse, new Label("JavaFX"));
39     Label lb5 = new Label("A pane inside a label", stackPane);
40     lb5.setContentDisplay(ContentDisplay.BOTTOM);
41
42     HBox pane = new HBox(20);
43     pane.getChildren().addAll(lb1, lb2, lb3, lb4, lb5);
44
45     // Create a scene and place it in the stage
46     Scene scene = new Scene(pane, 450, 150);
47     primaryStage.setTitle("LabelWithGraphic"); // Set the stage title
48     primaryStage.setScene(scene); // Place the scene in the stage
49     primaryStage.show(); // Display the stage
50   }
51 }
```

create a label

add labels to pane

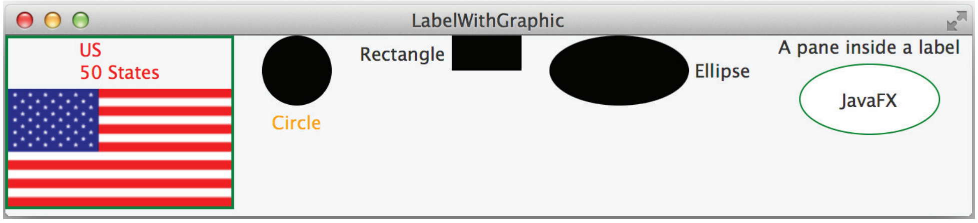


FIGURE 16.4 The program displays labels with texts and nodes. *Source*: booka/Fotolia.

The program creates a label with a text and an image (line 19). The text is `US\n50 States`, so it is displayed in two lines. Line 21 specifies that the image is placed at the bottom of the text.

The program creates a label with a text and a circle (line 24). The circle is placed on top of the text (line 25). The program creates a label with a text and a rectangle (line 28). The rectangle is placed on the right of the text (line 29). The program creates a label with a text and an ellipse (line 31). The ellipse is placed on the left of the text (line 32).

The program creates an ellipse (line 34), places it along with a label to a stack pane (line 38), and creates a label with a text and the stack pane as the node (line 39). As seen from this example, you can place any node in a label.

The program creates an `HBox` (line 42) and places all five labels into the `HBox` (line 43).

Check Point

16.2.1 How do you create a label with a node without a text?

16.2.2 How do you place a text on the right of the node in a label?

16.2.3 Can you display multiple lines of text in a label?

16.2.4 Can the text in a label be underlined?

16.3 Button

Key Point

A button is a control that triggers an action event when clicked.

JavaFX provides regular buttons, toggle buttons, check box buttons, and radio buttons. The common features of these buttons are defined in `ButtonBase` and `Labeled` classes as shown in Figure 16.5.

The `Labeled` class defines the common properties for labels and buttons. A button is just like a label, except that the button has the `onAction` property defined in the `ButtonBase` class, which sets a handler for handling a button's action.

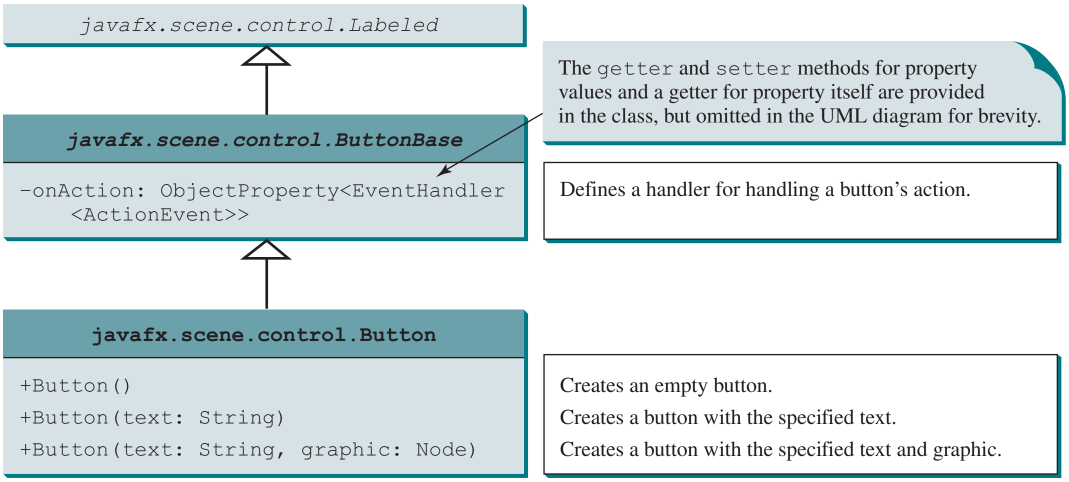

javafx.scene.control.Labeled
javafx.scene.control.ButtonBase
-onAction: ObjectProperty<EventHandler <ActionEvent>>
javafx.scene.control.Button
+Button()
+Button(text: String)
+Button(text: String, graphic: Node)
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
Defines a handler for handling a button's action.
Creates an empty button.
Creates a button with the specified text.
Creates a button with the specified text and graphic.

FIGURE 16.5 `ButtonBase` extends `Labeled` and defines common features for all buttons.

Listing 16.2 gives a program that uses the buttons to control the movement of a text, as shown in Figure 16.6.

LISTING 16.2 ButtonDemo.java

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;

public class ButtonDemo extends Application {
  protected Text text = new Text(50, 50, "JavaFX Programming");

  protected BorderPane getPane() {
    HBox paneForButtons = new HBox(20);
    Button btLeft = new Button("Left",
      new ImageView("image/left.gif"));
    Button btRight = new Button("Right",
      new ImageView("image/right.gif"));
    paneForButtons.getChildren().addAll(btLeft, btRight);
    paneForButtons.setAlignment(Pos.CENTER);
    paneForButtons.setStyle("-fx-border-color: green");

    BorderPane pane = new BorderPane();
    pane.setBottom(paneForButtons);

    Pane paneForText = new Pane();
    paneForText.getChildren().add(text);
    pane.setCenter(paneForText);

    btLeft.setOnAction(e -> text.setX(text.getX() - 10));
    btRight.setOnAction(e -> text.setX(text.getX() + 10));

    return pane;
```

create a button
add buttons to pane
create a border pane
add buttons to the bottom
add an action handler
return a pane

set pane to scene

```
36    }
37
38    @Override // Override the start method in the Application class
39    public void start(Stage primaryStage) {
40      // Create a scene and place it in the stage
41      Scene scene = new Scene(getPane(), 450, 200);
42      primaryStage.setTitle("ButtonDemo"); // Set the stage title
43      primaryStage.setScene(scene); // Place the scene in the stage
44      primaryStage.show(); // Display the stage
45    }
46  }
```

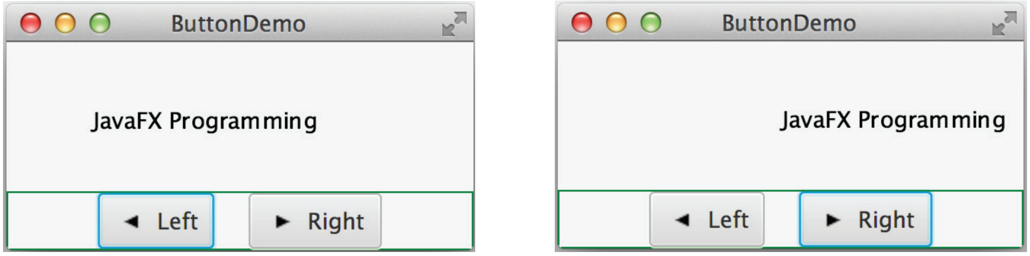


FIGURE 16.6 The program demonstrates using buttons. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

The program creates two buttons, `btLeft` and `btRight`, with each button containing a text and an image (lines 17–20). The buttons are placed in an `HBox` (line 21) and the `HBox` is placed in the bottom of a border pane (line 26). A text is created in line 13 and is placed in the center of the border pane (line 30). The action handler for `btLeft` moves the text to the left (line 32). The action handler for `btRight` moves the text to the right (line 33).

`getPane()` protected

The program purposely defines a protected `getPane()` method to return a pane (line 15). This method will be overridden by subclasses in the upcoming examples to add more nodes in the pane. The text is declared protected so it can be accessed by subclasses (line 13).

Check Point

16.3.1 How do you create a button with a text and a node? Can you apply all the methods for `Labeled` to `Button`?

16.3.2 Why is the `getPane()` method protected in Listing 16.2? Why is the data field `text` protected?

16.3.3 How do you set a handler for processing a button-clicked action?

16.4 CheckBox

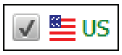
Key Point

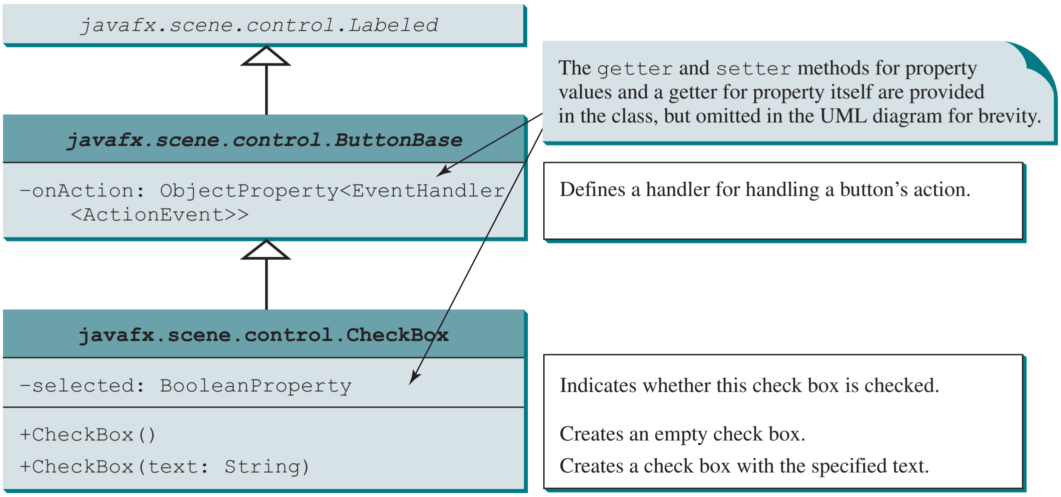
A `CheckBox` is used for the user to make a selection.

Like `Button`, `CheckBox` inherits all the properties such as `onAction`, `text`, `graphic`, `alignment`, `graphicTextGap`, `textFill`, and `contentDisplay` from `ButtonBase` and `Labeled`, as shown in Figure 16.7. In addition, it provides the `selected` property to indicate whether a check box is selected.

Here is an example of a check box with text `US`, a graphic image, green text color, black border, and initially selected.

```
CheckBox chkUS = new CheckBox("US");
chkUS.setGraphic(new ImageView("image/usIcon.gif"));
chkUS.setTextFill(Color.GREEN);
chkUS.setContentDisplay(ContentDisplay.LEFT);
chkUS.setStyle("-fx-border-color: black");
chkUS.setSelected(true);
chkUS.setPadding(new Insets(5, 5, 5, 5));
```




javafx.scene.control.Labeled
javafx.scene.control.ButtonBase
-onAction: ObjectProperty<EventHandler<ActionEvent>>
javafx.scene.control.CheckBox
-selected: BooleanProperty
+CheckBox()
+CheckBox(text: String)

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Defines a handler for handling a button's action.

Indicates whether this check box is checked.

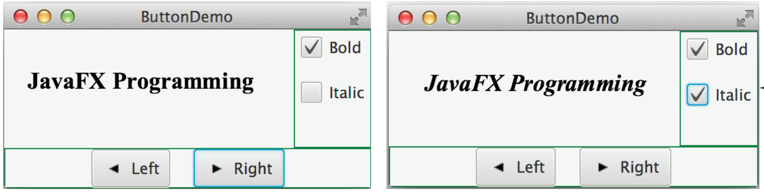
Creates an empty check box.

Creates a check box with the specified text.

FIGURE 16.7 **`CheckBox`** contains the properties inherited from **`ButtonBase`** and **`Labeled`**.

When a check box is clicked (checked or unchecked), it fires an **`ActionEvent`**. To see if a check box is selected, use the **`isSelected()`** method.

We now write a program that adds two check boxes named Bold and Italic to the preceding example to let the user specify whether the message is in bold or italic, as shown in Figure 16.8.


ButtonDemo
JavaFX Programming
☑ Bold
☐ Italic
◄ Left
► Right
ButtonDemo
JavaFX Programming
☑ Bold
☑ Italic
◄ Left
► Right

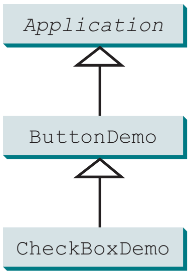
`VBox` containing two check boxes

FIGURE 16.8 The program demonstrates check boxes. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

There are at least two approaches to writing this program. The first is to revise the preceding **`ButtonDemo`** class to insert the code for adding the check boxes and processing their events. The second is to define a subclass that extends **`ButtonDemo`**. Please implement the first approach as an exercise. Listing 16.3 gives the code to implement the second approach.

LISTING 16.3 CheckBoxDemo.java

```java
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.control.CheckBox;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;

public class CheckBoxDemo extends ButtonDemo {
  @Override // Override the getPane() method in the super class
  protected BorderPane getPane() {
    BorderPane pane = super.getPane();

    Font fontBoldItalic = Font.font("Times New Roman",
      FontWeight.BOLD, FontPosture.ITALIC, 20);
```


Application
ButtonDemo
CheckBoxDemo

override `getPane()`

invoke `super.getPane()`

create fonts

pane for check boxes (line 27)

create check boxes (line 30)

create a handler (line 35)

set handler for action (line 50)

return a pane (line 53)

```
18      Font fontBold = Font.font("Times New Roman",
19        FontWeight.BOLD, FontPosture.REGULAR, 20);
20      Font fontItalic = Font.font("Times New Roman",
21        FontWeight.NORMAL, FontPosture.ITALIC, 20);
22      Font fontNormal = Font.font("Times New Roman",
23        FontWeight.NORMAL, FontPosture.REGULAR, 20);
24
25      text.setFont(fontNormal);
26
27      VBox paneForCheckBoxes = new VBox(20);
28      paneForCheckBoxes.setPadding(new Insets(5, 5, 5, 5));
29      paneForCheckBoxes.setStyle("-fx-border-color: green");
30      CheckBox chkBold = new CheckBox("Bold");
31      CheckBox chkItalic = new CheckBox("Italic");
32      paneForCheckBoxes.getChildren().addAll(chkBold, chkItalic);
33      pane.setRight(paneForCheckBoxes);
34
35      EventHandler<ActionEvent> handler = e -> {
36        if (chkBold.isSelected() && chkItalic.isSelected()) {
37          text.setFont(fontBoldItalic); // Both check boxes checked
38        }
39        else if (chkBold.isSelected()) {
40          text.setFont(fontBold); // The Bold check box checked
41        }
42        else if (chkItalic.isSelected()) {
43          text.setFont(fontItalic); // The Italic check box checked
44        }
45        else {
46          text.setFont(fontNormal); // Both check boxes unchecked
47        }
48      };
49
50      chkBold.setOnAction(handler);
51      chkItalic.setOnAction(handler);
52
53      return pane; // Return a new pane
54    }
55
56    public static void main(String[] args) {
57      launch(args);
58    }
59  }
```

`CheckBoxDemo` extends `ButtonDemo` and overrides the `getPane()` method (line 13). The new `getPane()` method invokes the `super.getPane()` method from the `ButtonDemo` class to obtain a border pane that contains the buttons and a text (line 14). The check boxes are created and added to `paneForCheckBoxes` (lines 30–32). `paneForCheckBoxes` is added to the border pane (lines 33).

The handler for processing the action event on check boxes is created in lines 35–48. It sets the appropriate font based on the status of the check boxes.

The `start` method for this JavaFX program is defined in `ButtonDemo` and inherited in `CheckBoxDemo`. Therefore, when you run `CheckBoxDemo`, the `start` method in `ButtonDemo` is invoked. Since the `getPane()` method is overridden in `CheckBoxDemo`, the method in `CheckBoxDemo` is invoked from line 41 in Listing 16.2, ButtonDemo.java. For additional information, see CheckPoint question 16.4.1.

Check Point

16.4.1 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
      Test test = new Test();
      test.new B().start();
    }

    class A {
      public void start() {
        System.out.println(getP());
      }

      public int getP() {
        return 1;
      }
    }

    class B extends A {
      public int getP() {
        return 2 + super.getP();
      }
    }
  }
```

16.4.2 How do you test if a check box is selected?

16.4.3 Can you apply all the methods for `Labeled` to `CheckBox`?

16.4.4 Can you set a node for the `graphic` property in a check box?

16.5 RadioButton

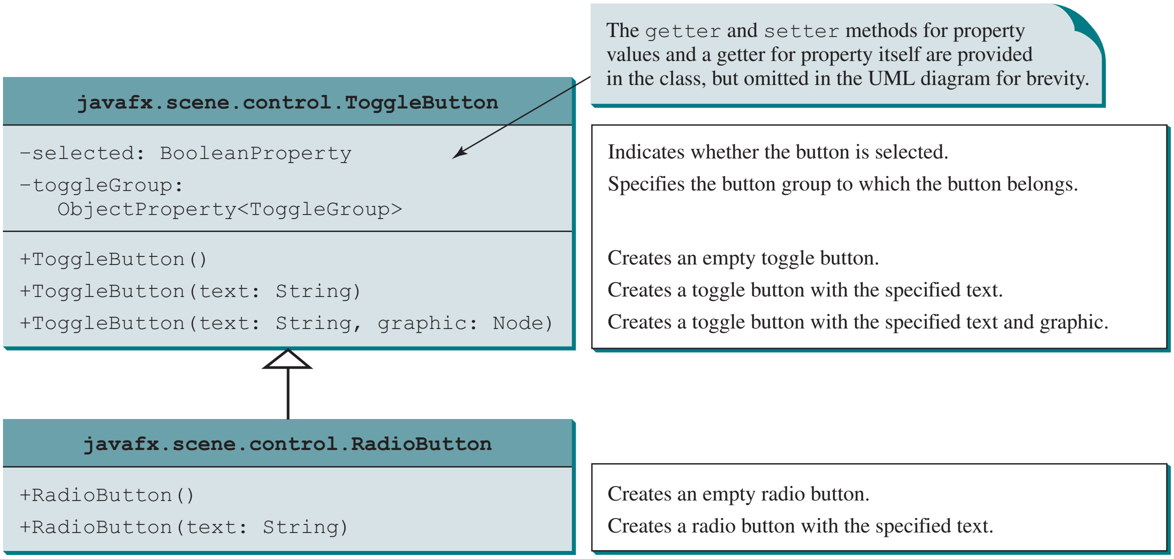
Radio buttons, also known as option buttons*, enable you to choose a single item from a group of choices.*

Key Point

In appearance, radio buttons resemble check boxes, but check boxes display a square that is either checked or blank, whereas radio buttons display a circle that is either filled (if selected) or blank (if not selected).

option buttons

`RadioButton` is a subclass of `ToggleButton`. The difference between a radio button and a toggle button is that a radio button displays a circle, but a toggle button is rendered similar to a button. The UML diagrams for `ToggleButton` and `RadioButton` are shown in Figure 16.9.


javafx.scene.control.ToggleButton
-selected: BooleanProperty
-toggleGroup: ObjectProperty<ToggleGroup>
+ToggleButton()
+ToggleButton(text: String)
+ToggleButton(text: String, graphic: Node)
javafx.scene.control.RadioButton
+RadioButton()
+RadioButton(text: String)

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Indicates whether the button is selected.
Specifies the button group to which the button belongs.

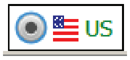
Creates an empty toggle button.
Creates a toggle button with the specified text.
Creates a toggle button with the specified text and graphic.

Creates an empty radio button.
Creates a radio button with the specified text.

FIGURE 16.9 `ToggleButton` and `RadioButton` are specialized buttons for making selections.

Here is an example of a radio button with text **`US`**, a graphic image, green text color, black border, and initially selected.

```
Radiobutton rbUS = new RadioButton("US");
rbUS.setGraphic(new ImageView("image/usIcon.gif"));
rbUS.setTextFill(Color.GREEN);
rbUS.setContentDisplay(ContentDisplay.LEFT);
rbUS.setStyle("-fx-border-color: black");
rbUS.setSelected(true);
rbUS.setPadding(new Insets(5, 5, 5, 5));
```



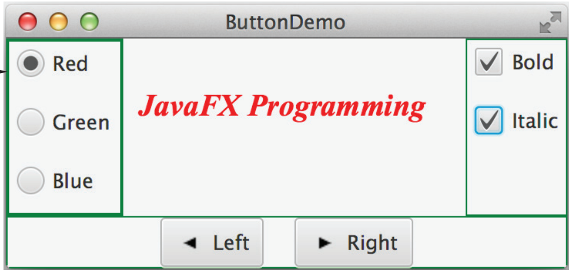
To group radio buttons, you need to create an instance of **`ToggleGroup`** and set a radio button's **`toggleGroup`** property to join the group, as follows:

```
ToggleGroup group = new ToggleGroup();
rbRed.setToggleGroup(group);
rbGreen.setToggleGroup(group);
rbBlue.setToggleGroup(group);
```

This code creates a button group for radio buttons **`rbRed`**, **`rbGreen`**, and **`rbBlue`** so buttons **`rbRed`**, **`rbGreen`**, and **`rbBlue`** are selected mutually exclusively. Without grouping, these buttons would be independent.

When a radio button is changed (selected or deselected), it fires an **`ActionEvent`**. To see if a radio button is selected, use the **`isSelected()`** method.

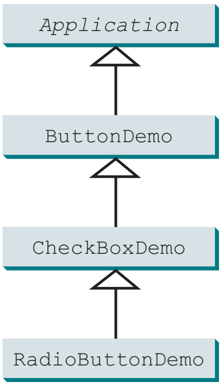
We now give a program that adds three radio buttons named Red, Green, and Blue to the preceding example to let the user choose the color of the message, as shown in Figure 16.10.



VBox containing three radio buttons

FIGURE 16.10 The program demonstrates using radio buttons. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Again, there are at least two approaches to writing this program. The first is to revise the preceding **`CheckBoxDemo`** class to insert the code for adding the radio buttons and processing their events. The second is to define a subclass that extends **`CheckBoxDemo`**. Listing 16.4 gives the code to implement the second approach.



LISTING 16.4 RadioButtonDemo.java

override `getPane()`
invoke `super.getPane()`

```
import javafx.geometry.Insets;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;

public class RadioButtonDemo extends CheckBoxDemo {
  @Override // Override the getPane() method in the super class
  protected BorderPane getPane() {
    BorderPane pane = super.getPane();

13      VBox paneForRadioButtons = new VBox(20);
14      paneForRadioButtons.setPadding(new Insets(5, 5, 5, 5));
15      paneForRadioButtons.setStyle
16          ("-fx-border-width: 2px; -fx-border-color: green");
17
18      RadioButton rbRed = new RadioButton("Red");
19      RadioButton rbGreen = new RadioButton("Green");
20      RadioButton rbBlue = new RadioButton("Blue");
21      paneForRadioButtons.getChildren().addAll(rbRed, rbGreen, rbBlue);
22      pane.setLeft(paneForRadioButtons);
23
24      ToggleGroup group = new ToggleGroup();
25      rbRed.setToggleGroup(group);
26      rbGreen.setToggleGroup(group);
27      rbBlue.setToggleGroup(group);
28
29      rbRed.setOnAction(e -> {
30        if (rbRed.isSelected()) {
31          text.setFill(Color.RED);
32        }
33      });
34
35      rbGreen.setOnAction(e -> {
36        if (rbGreen.isSelected()) {
37          text.setFill(Color.GREEN);
38        }
39      });
40
41      rbBlue.setOnAction(e -> {
42        if (rbBlue.isSelected()) {
43          text.setFill(Color.BLUE);
44        }
45      });
46
47      return pane;
48    }
49
50    public static void main(String[] args) {
51      launch(args);
52    }
53  }
```

pane for radio buttons

create radio buttons

add to border pane

group radio buttons

handle radio button

return border pane

RadioButtonDemo extends **CheckBoxDemo** and overrides the **getPane()** method (line 10). The new **getPane()** method invokes the **getPane()** method from the **CheckBoxDemo** class to create a border pane that contains the check boxes, buttons, and a text (line 11). This border pane is returned from invoking **super.getPane()**. The radio buttons are created and added to **paneForRadioButtons** (lines 18–21). **paneForRadioButtons** is added to the border pane (line 22).

The radio buttons are grouped together in lines 24–27. The handlers for processing the action event on radio buttons are created in lines 29–45. It sets the appropriate color based on the status of the radio buttons.

The **start** method for this JavaFX program is defined in **ButtonDemo** and inherited in **CheckBoxDemo** then in **RadioButtonDemo**. Thus, when you run **RadioButtonDemo**, the **start** method in **ButtonDemo** is invoked. Since the **getPane()** method is overridden in **RadioButtonDemo**, the method in **RadioButtonDemo** is invoked from line 41 in Listing 16.2, ButtonDemo.java.

Check Point

16.5.1 How do you test if a radio button is selected?

16.5.2 Can you apply all the methods for **Labeled** to **RadioButton**?

16.5.3 Can you set any node in the **graphic** property in a radio button?

16.5.4 How do you group radio buttons?`

16.6 TextField

Key Point *A text field can be used to enter or display a string.*

`TextField` is a subclass of **`TextInputControl`**. Figure 16.11 lists the properties and constructors in **`TextField`**.

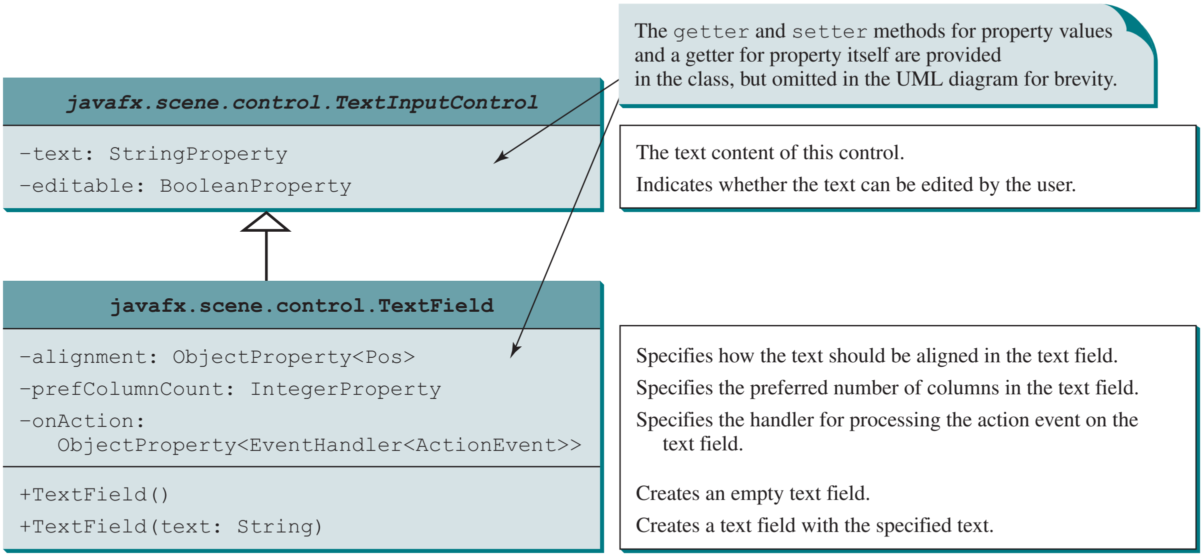
Here is an example of creating a noneditable text field with red text color, a specified font, and right horizontal alignment:

```
TextField tfMessage = new TextField("T-Storm");
tfMessage.setEditable(false);
tfMessage.setStyle("-fx-text-fill: red");
tfMessage.setFont(Font.font("Times", 20));
tfMessage.setAlignment(Pos.BASELINE_RIGHT);
```

T-Storm

When you move the cursor in the text field and press the *Enter* key, it fires an **`ActionEvent`**.

Listing 16.5 gives a program that adds a text field to the preceding example to let the user set a new message, as shown in Figure 16.12.


The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

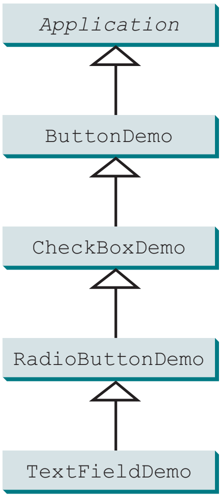
javafx.scene.control.TextInputControl

`-text: StringProperty`	The text content of this control.
`-editable: BooleanProperty`	Indicates whether the text can be edited by the user.

javafx.scene.control.TextField

`-alignment: ObjectProperty<Pos>`	Specifies how the text should be aligned in the text field.
`-prefColumnCount: IntegerProperty`	Specifies the preferred number of columns in the text field.
`-onAction: ObjectProperty<EventHandler<ActionEvent>>`	Specifies the handler for processing the action event on the text field.
`+TextField()`	Creates an empty text field.
`+TextField(text: String)`	Creates a text field with the specified text.

FIGURE 16.11 **`TextField`** enables the user to enter or display a string.



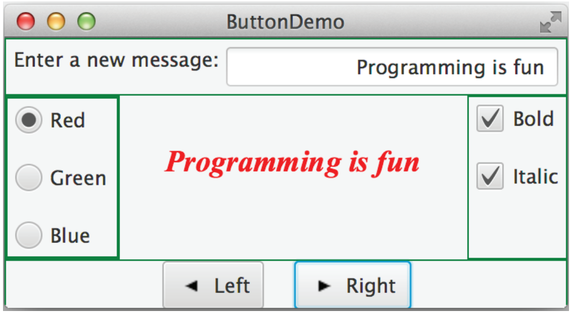

Enter a new message: Programming is fun

FIGURE 16.12 The program demonstrates using text fields. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Listing 16.5 TextFieldDemo.java

```
 1  import javafx.geometry.Insets;
 2  import javafx.geometry.Pos;
 3  import javafx.scene.control.Label;
 4  import javafx.scene.control.TextField;
 5  import javafx.scene.layout.BorderPane;
 6
 7  public class TextFieldDemo extends RadioButtonDemo {
 8    @Override // Override the getPane() method in the super class
 9    protected BorderPane getPane() {
10      BorderPane pane = super.getPane();
11
12      BorderPane paneForTextField = new BorderPane();
13      paneForTextField.setPadding(new Insets(5, 5, 5, 5));
14      paneForTextField.setStyle("-fx-border-color: green");
15      paneForTextField.setLeft(new Label("Enter a new message: "));
16
17      TextField tf = new TextField();
18      tf.setAlignment(Pos.BOTTOM_RIGHT);
19      paneForTextField.setCenter(tf);
20      pane.setTop(paneForTextField);
21
22      tf.setOnAction(e -> text.setText(tf.getText()));
23
24      return pane;
25    }
26
27    public static void main(String[] args) {
28      launch(args);
29    }
30  }
```

override `getPane()`
invoke `super.getPane()`
pane for label and text field
create text field
add to border pane
handle text field action
return border pane

`TextFieldDemo` extends **`RadioButtonDemo`** (line 7) and adds a label and a text field to let the user enter a new text (lines 12–20). After you set a new text in the text field and press the *Enter* key, a new message is displayed (line 22). Pressing the *Enter* key on the text field triggers an action event.

Note

If a text field is used for entering a password, use **`PasswordField`** to replace **`TextField`**. **`PasswordField`** extends **`TextField`** and hides the input text with echo characters **`******`**.

`PasswordField`

Check Point

16.6.1 Can you disable editing of a text field?

16.6.2 Can you apply all the methods for **`TextInputControl`** to **`TextField`**?

16.6.3 Can you set a node as the **`graphic`** property in a text field?

16.6.4 How do you align the text in a text field to the right?

16.7 TextArea

*A **`TextArea`** enables the user to enter multiple lines of text.*

Key Point

If you want to let the user enter multiple lines of text, you may create several instances of **`TextField`**. A better alternative, however, is to use **`TextArea`**, which enables the user to enter multiple lines of text. Figure 16.13 lists the properties and constructors in **`TextArea`**.

Here is an example of creating a text area with **`5`** rows and **`20`** columns, wrapped to the next line, **`red`** text color, and **`Courier`** font **`20`** pixels.

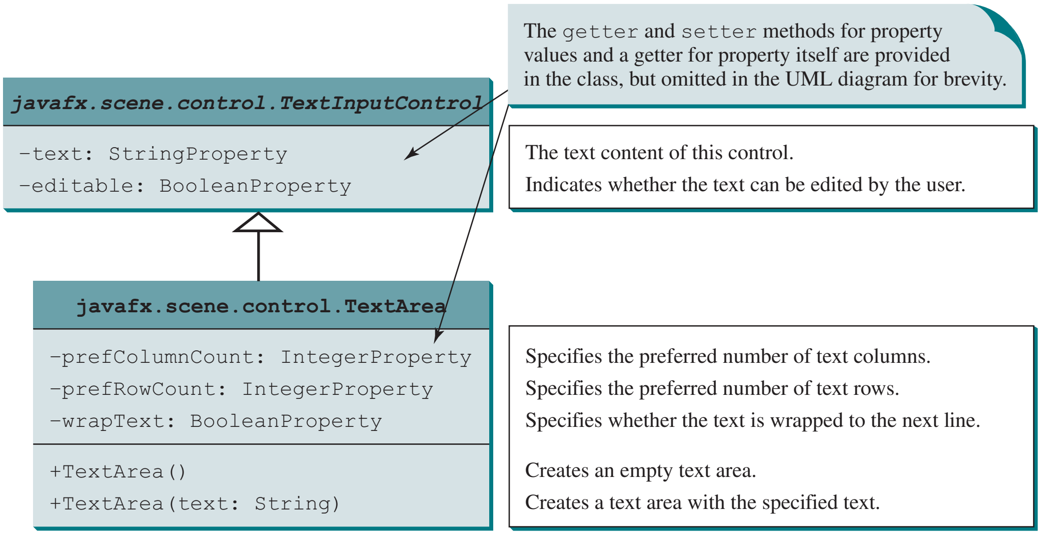

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
javafx.scene.control.TextInputControl
-text: StringProperty
-editable: BooleanProperty
The text content of this control.
Indicates whether the text can be edited by the user.
javafx.scene.control.TextArea
-prefColumnCount: IntegerProperty
-prefRowCount: IntegerProperty
-wrapText: BooleanProperty
+TextArea()
+TextArea(text: String)
Specifies the preferred number of text columns.
Specifies the preferred number of text rows.
Specifies whether the text is wrapped to the next line.
Creates an empty text area.
Creates a text area with the specified text.

Figure 16.13 `TextArea` enables the user to enter or display multiple lines of characters.

```
TextArea taNote = new TextArea("This is a text area");
taNote.setPrefColumnCount(20);
taNote.setPrefRowCount(5);
taNote.setWrapText(true);
taNote.setStyle("-fx-text-fill: red");
taNote.setFont(Font.font("Times", 20));
```

`TextArea` provides scrolling, but often it is useful to create a `ScrollPane` object to hold an instance of `TextArea` and let `ScrollPane` handle scrolling for `TextArea`, as follows:

```
// Create a scroll pane to hold text area
ScrollPane scrollPane = new ScrollPane(taNote);
```

`ScrollPane`

Tip

You can place any node in a `ScrollPane`. `ScrollPane` automatically provides vertical and horizontal scrolling if the node is too large to fit in the viewing area.

We now give a program that displays an image and a short text in a label, and a long text in a text area, as shown in Figure 16.14.

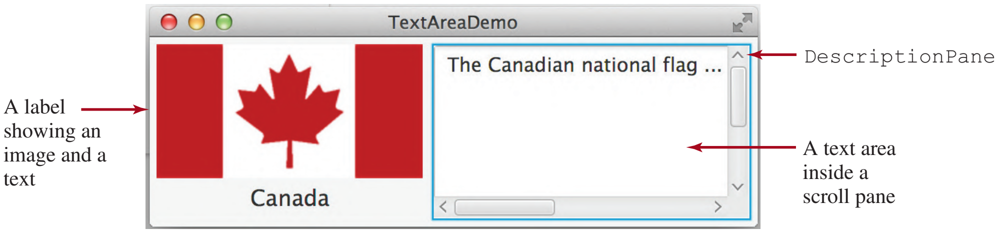

TextAreaDemo
The Canadian national flag ...
Canada
A label showing an image and a text
DescriptionPane
A text area inside a scroll pane

Figure 16.14 The program displays an image in a label, a title in a label, and text in the text area. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Here are the major steps in the program:

1. Define a class named `DescriptionPane` that extends `BorderPane`, as shown in Listing 16.6. This class contains a text area inside a scroll pane and a label for displaying an image icon and a title. The class `DescriptionPane` will be reused in later examples.

2. Define a class named **TextAreaDemo** that extends **Application**, as shown in Listing 16.7. Create an instance of **DescriptionPane** and add it to the scene. The relationship between **DescriptionPane** and **TextAreaDemo** is shown in Figure 16.15.

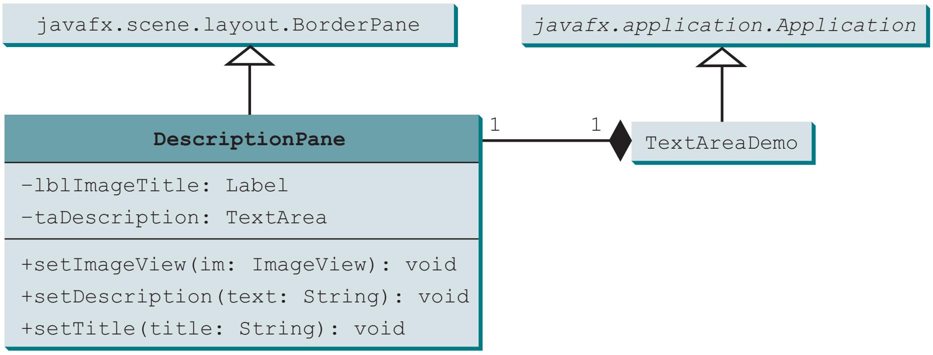

javafx.scene.layout.BorderPane
javafx.application.Application
DescriptionPane
1
1
TextAreaDemo
-lblImageTitle: Label
-taDescription: TextArea
+setImageView(im: ImageView): void
+setDescription(text: String): void
+setTitle(title: String): void

FIGURE 16.15 **TextAreaDemo** uses **DescriptionPane** to display an image, title, and text description of a national flag.

LISTING 16.6 DescriptionPane.java

```java
import javafx.geometry.Insets;
import javafx.scene.control.Label;
import javafx.scene.control.ContentDisplay;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.scene.text.Font;

public class DescriptionPane extends BorderPane {
  /** Label for displaying an image and a title */
  private Label lblImageTitle = new Label();

  /** Text area for displaying text */
  private TextArea taDescription = new TextArea();

  public DescriptionPane() {
    // Center the icon and text and place the text under the icon
    lblImageTitle.setContentDisplay(ContentDisplay.TOP);
    lblImageTitle.setPrefSize(200, 100);

    // Set the font in the label and the text field
    lblImageTitle.setFont(new Font("SansSerif", 16));
    taDescription.setFont(new Font("Serif", 14));

    taDescription.setWrapText(true);
    taDescription.setEditable(false);

    // Create a scroll pane to hold the text area
    ScrollPane scrollPane = new ScrollPane(taDescription);

    // Place label and scroll pane in the border pane
    setLeft(lblImageTitle);
    setCenter(scrollPane);
    setPadding(new Insets(5, 5, 5, 5));
  }

  /** Set the title */
```

label

text area

label properties

wrap text

read only

scroll pane

```
39    public void setTitle(String title) {
40      lblImageTitle.setText(title);
41    }
42
43    /** Set the image view */
44    public void setImageView(ImageView icon) {
45      lblImageTitle.setGraphic(icon);
46    }
47
48    /** Set the text description */
49    public void setDescription(String text) {
50      taDescription.setText(text);
51    }
52  }
```

The text area is inside a `ScrollPane` (line 30), which provides scrolling functions for the text area.

The `wrapText` property is set to `true` (line 26) so the line is automatically wrapped when the text cannot fit in one line. The text area is set as noneditable (line 27), so you cannot edit the description in the text area.

It is not necessary to define a separate class for `DescriptionPane` in this example. However, this class was defined for reuse in the next section, where you will use it to display a description pane for various images.

LISTING 16.7 TextAreaDemo.java

create `descriptionPane`

set title

set image

add `descriptionPane` to scene

```
 1  import javafx.application.Application;
 2  import javafx.stage.Stage;
 3  import javafx.scene.Scene;
 4  import javafx.scene.image.ImageView;
 5
 6  public class TextAreaDemo extends Application {
 7    @Override // Override the start method in the Application class
 8    public void start(Stage primaryStage) {
 9      // Declare and create a description pane
10      DescriptionPane descriptionPane = new DescriptionPane();
11
12      // Set title, text, and image in the description pane
13      descriptionPane.setTitle("Canada");
14      String description = "The Canadian national flag ... ";
15      descriptionPane.setImageView(new ImageView("image/ca.gif"));
16      descriptionPane.setDescription(description);
17
18      // Create a scene and place it in the stage
19      Scene scene = new Scene(descriptionPane, 450, 200);
20      primaryStage.setTitle("TextAreaDemo"); // Set the stage title
21      primaryStage.setScene(scene); // Place the scene in the stage
22      primaryStage.show(); // Display the stage
23    }
24  }
```

The program creates an instance of `DescriptionPane` (line 10) and sets the title (line 13), image (line 15), and text (line 16) in the description pane. `DescriptionPane` is a subclass of `Pane`. `DescriptionPane` contains a label for displaying an image, a title, and a text area for displaying a description of the image.

Check Point

16.7.1 How do you create a text area with `10` rows and `20` columns?

16.7.2 How do you obtain the text from a text area?

16.7.3 Can you disable editing of a text area?

16.7.4 What method do you use to wrap text to the next line in a text area?

16.8 ComboBox

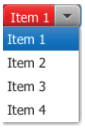
A combo box, also known as a choice list or drop-down list, contains a list of items from which the user can choose.

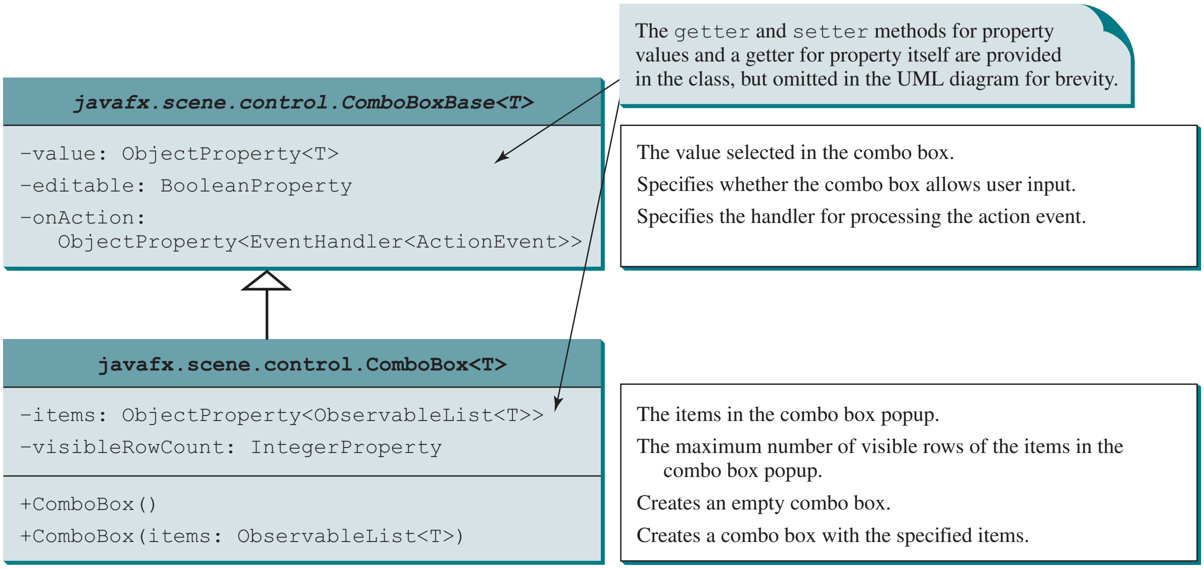
Key Point

A combo box is useful for limiting a user's range of choices and avoids the cumbersome validation of data input. Figure 16.16 lists several frequently used properties and constructors in `ComboBox`. `ComboBox` is defined as a generic class like the `ArrayList` class. The generic type `T` specifies the element type for the elements stored in a combo box.

The following statements create a combo box with four items, red color, and value set to the first item:

```
ComboBox<String> cbo = new ComboBox<>();
cbo.getItems().addAll("Item 1", "Item 2",
  "Item 3", "Item 4");
cbo.setStyle("-fx-color: #EB0D1B");
cbo.setValue("item 1");
```





The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.control.ComboBoxBase<T>

-value: ObjectProperty<T>	The value selected in the combo box.
-editable: BooleanProperty	Specifies whether the combo box allows user input.
-onAction: ObjectProperty<EventHandler<ActionEvent>>	Specifies the handler for processing the action event.

javafx.scene.control.ComboBox<T>

-items: ObjectProperty<ObservableList<T>>	The items in the combo box popup.
-visibleRowCount: IntegerProperty	The maximum number of visible rows of the items in the combo box popup.
+ComboBox()	Creates an empty combo box.
+ComboBox(items: ObservableList<T>)	Creates a combo box with the specified items.

Figure 16.16 `ComboBox` enables the user to select an item from a list of items.

`ComboBox` inherits from `ComboBoxBase`. `ComboBox` can fire an `ActionEvent`. Whenever an item is selected, an `ActionEvent` is fired. `ObservableList` is a subinterface of `java.util.List`. Therefore, you can apply all the methods defined in `List` for an `ObservableList`. For convenience, JavaFX provides the static method `FXCollections.observableArrayList(arrayOfElements)` for creating an `ObservableList` from an array of elements.

Listing 16.8 gives a program that lets the user view an image and a description of a country's flag by selecting the country from a combo box, as shown in Figure 16.17.

Here are the major steps in the program:

1. Create the user interface.
   Create a combo box with country names as its selection values. Create a `DescriptionPane` object (the `DescriptionPane` class was introduced in the preceding section). Place the combo box at the top of the border pane, and the description pane in the center of the border pane.

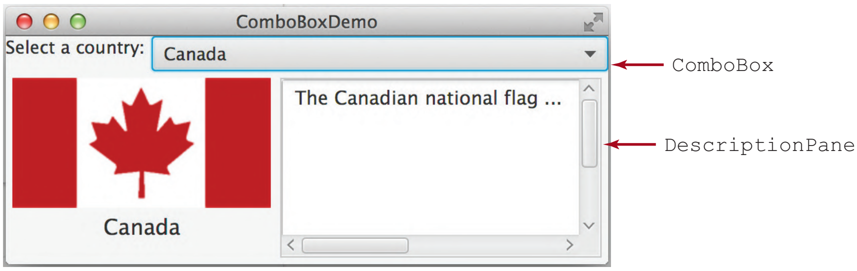


FIGURE 16.17 Information about a country, including an image and a description of its flag, is displayed when the country is selected in the combo box. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

2. Process the event.
   Create a handler for handling action event from the combo box to set the flag title, image, and text in the description pane for the selected country name.

LISTING 16.8 ComboBoxDemo.java

countries

image views

description

combo box

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;

public class ComboBoxDemo extends Application {
  // Declare an array of Strings for flag titles
  private String[] flagTitles = {"Canada", "China", "Denmark",
    "France", "Germany", "India", "Norway", "United Kingdom",
    "United States of America"};

  // Declare an ImageView array for the national flags of 9 countries
  private ImageView[] flagImage = {new ImageView("image/ca.gif"),
    new ImageView("image/china.gif"),
    new ImageView("image/denmark.gif"),
    new ImageView("image/fr.gif"),
    new ImageView("image/germany.gif"),
    new ImageView("image/india.gif"),
    new ImageView("image/norway.gif"),
    new ImageView("image/uk.gif"), new ImageView("image/us.gif")};

  // Declare an array of strings for flag descriptions
  private String[] flagDescription = new String[9];

  // Declare and create a description pane
  private DescriptionPane descriptionPane = new DescriptionPane();

  // Create a combo box for selecting countries
  private ComboBox<String> cbo = new ComboBox<>(); // flagTitles;

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Set text description
39       flagDescription[0] = "The Canadian national flag ... ";
40       flagDescription[1] = "Description for China ... ";
41       flagDescription[2] = "Description for Denmark ... ";
42       flagDescription[3] = "Description for France ... ";
43       flagDescription[4] = "Description for Germany ... ";
44       flagDescription[5] = "Description for India ... ";
45       flagDescription[6] = "Description for Norway ... ";
46       flagDescription[7] = "Description for UK ... ";
47       flagDescription[8] = "Description for US ... ";
48
49       // Set the first country (Canada) for display
50       setDisplay(0);
51
52       // Add combo box and description pane to the border pane
53       BorderPane pane = new BorderPane();
54
55       BorderPane paneForComboBox = new BorderPane();
56       paneForComboBox.setLeft(new Label("Select a country: "));
57       paneForComboBox.setCenter(cbo);
58       pane.setTop(paneForComboBox);
59       cbo.setPrefWidth(400);
60       cbo.setValue("Canada");
61
62       ObservableList<String> items =
63         FXCollections.observableArrayList(flagTitles);
64       cbo.getItems().addAll(items);
65       pane.setCenter(descriptionPane);
66
67       // Display the selected country
68       cbo.setOnAction(e -> setDisplay(items.indexOf(cbo.getValue())));
69
70       // Create a scene and place it in the stage
71       Scene scene = new Scene(pane, 450, 170);
72       primaryStage.setTitle("ComboBoxDemo"); // Set the stage title
73       primaryStage.setScene(scene); // Place the scene in the stage
74       primaryStage.show(); // Display the stage
75     }
76
77     /** Set display information on the description pane */
78     public void setDisplay(int index) {
79       descriptionPane.setTitle(flagTitles[index]);
80       descriptionPane.setImageView(flagImage[index]);
81       descriptionPane.setDescription(flagDescription[index]);
82     }
83   }
```

set combo box value

observable list

add to combo box

The program stores the flag information in three arrays: **`flagTitles`**, **`flagImage`**, and **`flagDescription`** (lines 13–28). The array **`flagTitles`** contains the names of nine countries, the array **`flagImage`** contains image views of each of the nine countries' flags, and the array **`flagDescription`** contains descriptions of the flags.

The program creates an instance of **`DescriptionPane`** (line 31), which was presented in Listing 16.6, DescriptionPane.java. The program creates a combo box with values from **`flagTitles`** (lines 62 and 63). The **`getItems()`** method returns a list from the combo box (line 64) and the **`addAll`** method adds multiple items into the list.

When the user selects an item in the combo box, the action event triggers the execution of the handler. The handler finds the selected index (line 68) and invokes the **`setDisplay(int index)`** method to set its corresponding flag title, flag image, and flag description on the pane (lines 78–82).

Check Point

16.8.1 How do you create a combo box and add three items to it?

16.8.2 How do you retrieve an item from a combo box? How do you retrieve a selected item from a combo box?

16.8.3 How do you get the number of items in a combo box? How do you retrieve an item at a specified index in a combo box?

16.8.4 What events would a `ComboBox` fire upon selecting a new item?

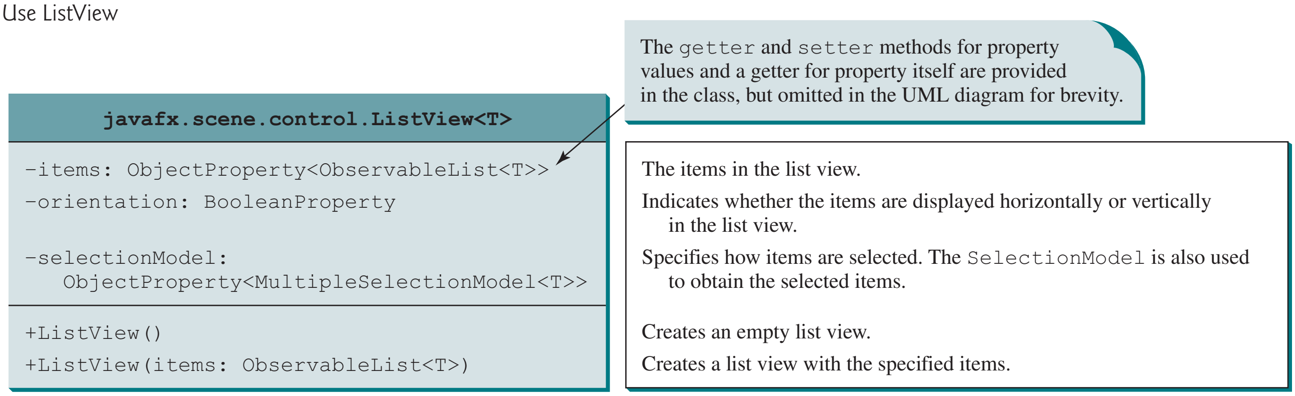
16.9 `ListView`

Key Point

A list view is a control that basically performs the same function as a combo box, but it enables the user to choose a single value or multiple values.

VideoNote
Use ListView

Figure 16.18 lists several frequently used properties and constructors in `ListView`. `ListView` is defined as a generic class like the `ArrayList` class. The generic type `T` specifies the element type for the elements stored in a list view.

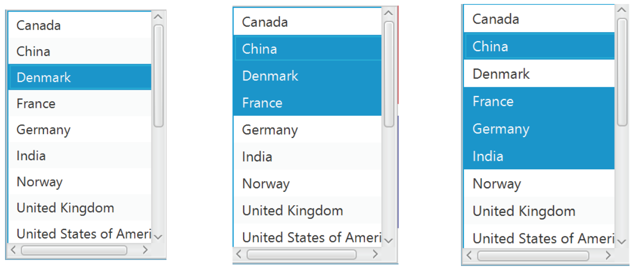


The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.control.ListView<T>	
-items: ObjectProperty<ObservableList<T>>	The items in the list view.
-orientation: BooleanProperty	Indicates whether the items are displayed horizontally or vertically in the list view.
-selectionModel: ObjectProperty<MultipleSelectionModel<T>>	Specifies how items are selected. The `SelectionModel` is also used to obtain the selected items.
+ListView()	Creates an empty list view.
+ListView(items: ObservableList<T>)	Creates a list view with the specified items.

FIGURE 16.18 `ListView` enables the user to select one or multiple items from a list of items.

The `getSelectionModel()` method returns an instance of `SelectionModel`, which contains the methods for setting a selection mode and obtaining selected indices and items. The selection mode is defined in one of the two constants `SelectionMode.MULTIPLE` and `SelectionMode.SINGLE`, which indicates whether a single item or multiple items can be selected. The default value is `SelectionMode.SINGLE`. Figure 16.19a shows a single selection and Figures 16.19b and c show multiple selections.



(a) Single selection (b) Multiple selection (c) Multiple selection

FIGURE 16.19 `SelectionMode` has two selection modes: single selection and multiple-interval selection. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

The following statements create a list view of six items with multiple selections allowed:

```
ObservableList<String> items =
  FXCollections.observableArrayList("Item 1", "Item 2",
    "Item 3", "Item 4", "Item 5", "Item 6");
ListView<String> lv = new ListView<>(items);
lv.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

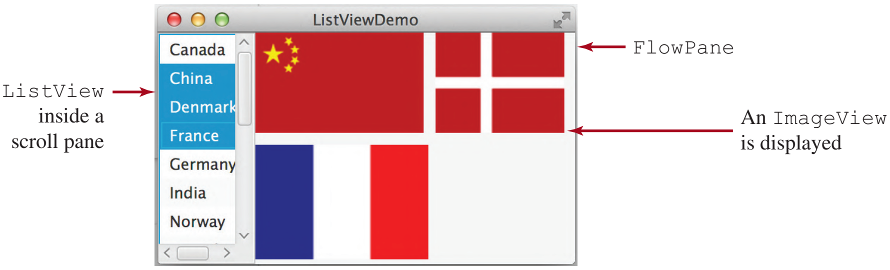
The selection model in a list view has the `selectedItemProperty` property, which is an instance of `Observable`. As discussed in Section 15.10, you can add a listener to this property for handling the property change as follows:

```
lv.getSelectionModel().selectedItemProperty().addListener(
  new InvalidationListener() {
    public void invalidated(Observable ov) {
      System.out.println("Selected indices: "
        + lv.getSelectionModel().getSelectedIndices());
      System.out.println("Selected items: "
        + lv.getSelectionModel().getSelectedItems());
    }
  });
```

This anonymous inner class can be simplified using a lambda expression as follows:

```
lv.getSelectionModel().selectedItemProperty().addListener(ov -> {
  System.out.println("Selected indices: "
    + lv.getSelectionModel().getSelectedIndices());
  System.out.println("Selected items: "
    + lv.getSelectionModel().getSelectedItems());
});
```

Listing 16.9 gives a program that lets users select the countries in a list and displays the flags of the selected countries in the image views. Figure 16.20 shows a sample run of the program.



`ListView` inside a scroll pane

`FlowPane`

An `ImageView` is displayed

FIGURE 16.20 When the countries in the list are selected, corresponding images of their flags are displayed in the image views. *Source*: booka/Fotolia.

Here are the major steps in the program:

1. Create the user interface.
   Create a list view with nine country names as selection values and place the list view inside a scroll pane. Place the scroll pane on the left of a border pane. Create nine image views to be used to display the countries' flag images. Create a flow pane to hold the image views and place the pane in the center of the border pane.

2. Process the event.
   Create a listener to implement the `invalidated` method in the `InvalidationListener` interface to place the selected countries' flag image views in the pane.

Listing 16.9 ListViewDemo.java

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.collections.FXCollections;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.SelectionMode;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.FlowPane;

public class ListViewDemo extends Application {
  // Declare an array of Strings for flag titles
  private String[] flagTitles = {"Canada", "China", "Denmark",
    "France", "Germany", "India", "Norway", "United Kingdom",
    "United States of America"};

  // Declare an ImageView array for the national flags of 9 countries
  private ImageView[] ImageViews = {
    new ImageView("image/ca.gif"),
    new ImageView("image/china.gif"),
    new ImageView("image/denmark.gif"),
    new ImageView("image/fr.gif"),
    new ImageView("image/germany.gif"),
    new ImageView("image/india.gif"),
    new ImageView("image/norway.gif"),
    new ImageView("image/uk.gif"),
    new ImageView("image/us.gif")
  };

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    ListView<String> lv = new ListView<>
      (FXCollections.observableArrayList(flagTitles));
    lv.setPrefSize(400, 400);
    lv.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);

    // Create a pane to hold image views
    FlowPane imagePane = new FlowPane(10, 10);
    BorderPane pane = new BorderPane();
    pane.setLeft(new ScrollPane(lv));
    pane.setCenter(imagePane);

    lv.getSelectionModel().selectedItemProperty().addListener(
      ov -> {
        imagePane.getChildren().clear();
        for (Integer i: lv.getSelectionModel().getSelectedIndices()) {
          imagePane.getChildren().add(ImageViews[i]);
        }
    });

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 450, 170);
    primaryStage.setTitle("ListViewDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

create a list view

set list view properties

place list view in pane

listen to item selected

add image views of selected items

The program creates an array of strings for countries (lines 14–16) and an array of nine image views for displaying flag images for nine countries (lines 19–29) in the same order as in the array of countries. The items in the list view are from the array of countries (line 34). Thus, the index `0` of the image view array corresponds to the first country in the list view.

The list view is placed in a scroll pane (line 41) so it can be scrolled when the number of items in the list extends beyond the viewing area.

By default, the selection mode of the list view is single. The selection mode for the list view is set to multiple (line 36), which allows the user to select multiple items in the list view. When the user selects countries in the list view, the listener's handler (lines 44–50) is executed, which gets the indices of the selected items and adds their corresponding image views to the flow pane.

Check Point

16.9.1 How do you create an observable list with an array of strings?

16.9.2 How do you set the orientation in a list view?

16.9.3 What selection modes are available for a list view? What is the default selection mode? How do you set a selection mode?

16.9.4 How do you obtain the selected items and selected indices?

16.10 ScrollBar

Key Point

`ScrollBar` *is a control that enables the user to select from a range of values.*

Figure 16.21 shows a scroll bar. Normally, the user changes the value of a scroll bar by making a gesture with the mouse. For example, the user can drag the scroll bar's thumb, click on the scroll bar track, or the scroll bar's left or right buttons.

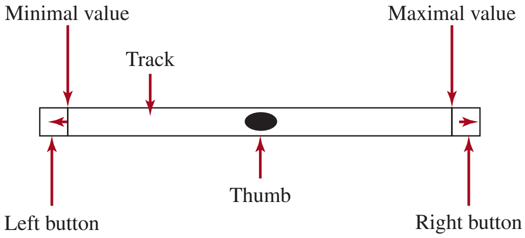
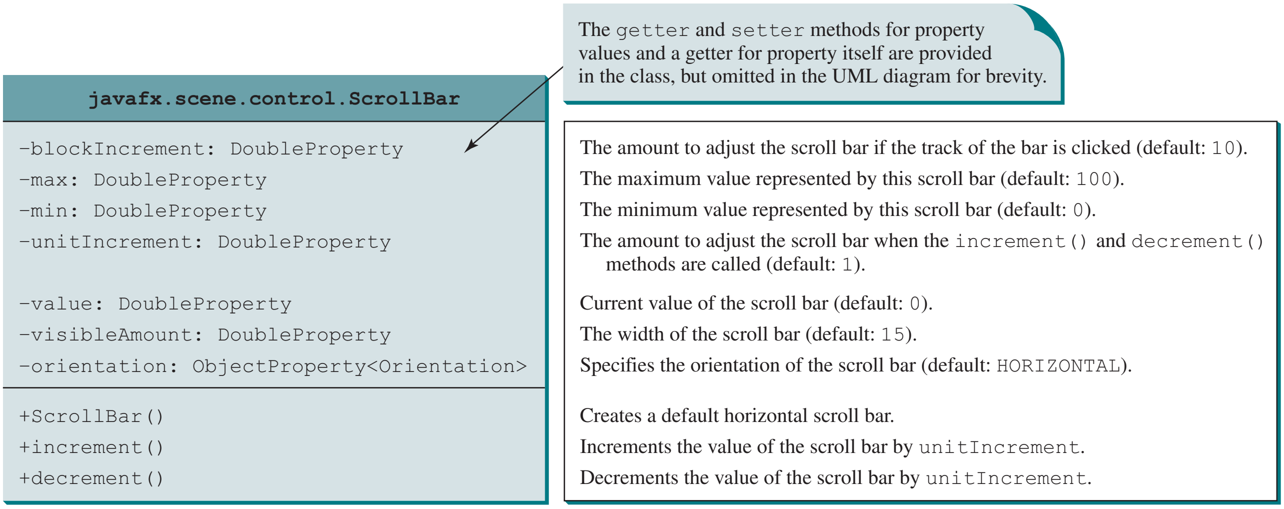


FIGURE 16.21 A scroll bar graphically represents a range of values.

`ScrollBar` has the following properties, as shown in Figure 16.22.



The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.control.ScrollBar	
`-blockIncrement: DoubleProperty`	The amount to adjust the scroll bar if the track of the bar is clicked (default: `10`).
`-max: DoubleProperty`	The maximum value represented by this scroll bar (default: `100`).
`-min: DoubleProperty`	The minimum value represented by this scroll bar (default: `0`).
`-unitIncrement: DoubleProperty`	The amount to adjust the scroll bar when the `increment()` and `decrement()` methods are called (default: `1`).
`-value: DoubleProperty`	Current value of the scroll bar (default: `0`).
`-visibleAmount: DoubleProperty`	The width of the scroll bar (default: `15`).
`-orientation: ObjectProperty<Orientation>`	Specifies the orientation of the scroll bar (default: `HORIZONTAL`).
`+ScrollBar()`	Creates a default horizontal scroll bar.
`+increment()`	Increments the value of the scroll bar by `unitIncrement`.
`+decrement()`	Decrements the value of the scroll bar by `unitIncrement`.

FIGURE 16.22 `ScrollBar` enables the user to select from a range of values.

Note

The width of the scroll bar's track corresponds to **`max + visibleAmount`**. When a scroll bar is set to its maximum value, the left side of the bubble is at **`max`**, and the right side is at **`max + visibleAmount`**.

When the user changes the value of the scroll bar, it notifies the listener of the change. You can register a listener on the scroll bar's `valueProperty` for responding to this change as follows:

```
ScrollBar sb = new ScrollBar();
sb.valueProperty().addListener(ov -> {
  System.out.println("old value: " + oldVal);
  System.out.println("new value: " + newVal);
});
```

Listing 16.10 gives a program that uses horizontal and vertical scroll bars to move a text displayed on a pane. The horizontal scroll bar is used to move the text to the left and the right, and the vertical scroll bar to move it up and down. A sample run of the program is shown in Figure 16.23.

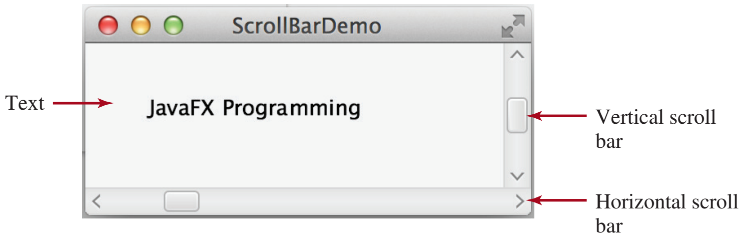


FIGURE 16.23 The scroll bars move the message on a pane horizontally and vertically. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Here are the major steps in the program:

1. Create the user interface.
   Create a `Text` object and place it in a pane and place the pane in the center of the border pane. Create a vertical scroll bar and place it on the right of the border pane. Create a horizontal scroll bar and place it at the bottom of the border pane.
2. Process the event.
   Create listeners to move the text according to the bar movement in the scroll bars upon the change of the `value` property.

LISTING 16.10 ScrollBarDemo.java

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.geometry.Orientation;
import javafx.scene.Scene;
import javafx.scene.control.ScrollBar;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;

public class ScrollBarDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    Text text = new Text(20, 20, "JavaFX Programming");

    ScrollBar sbHorizontal = new ScrollBar();
    ScrollBar sbVertical = new ScrollBar();
    sbVertical.setOrientation(Orientation.VERTICAL);

    // Create a text in a pane
```

horizontal scroll bar
vertical scroll bar

```
20      Pane paneForText = new Pane();
21      paneForText.getChildren().add(text);
22
23      // Create a border pane to hold text and scroll bars
24      BorderPane pane = new BorderPane();
25      pane.setCenter(paneForText);
26      pane.setBottom(sbHorizontal);
27      pane.setRight(sbVertical);
28
29      // Listener for horizontal scroll bar value change
30      sbHorizontal.valueProperty().addListener(ov ->
31        text.setX(sbHorizontal.getValue() * paneForText.getWidth() /
32          sbHorizontal.getMax()));
33
34      // Listener for vertical scroll bar value change
35      sbVertical.valueProperty().addListener(ov ->
36        text.setY(sbVertical.getValue() * paneForText.getHeight() /
37          sbVertical.getMax()));
38
39      // Create a scene and place it in the stage
40      Scene scene = new Scene(pane, 450, 170);
41      primaryStage.setTitle("ScrollBarDemo"); // Set the stage title
42      primaryStage.setScene(scene); // Place the scene in the stage
43      primaryStage.show(); // Display the stage
44    }
45  }
```

add text to a pane

border pane

set new location for text

set new location for text

The program creates a text (line 13) and two scroll bars (**`sbHorizontal`** and **`sbVertical`**) (lines 15 and 16). The text is placed in a pane (line 21) that is then placed in the center of the border pane (line 25). If the text were directly placed in the center of the border pane, the position of the text could not be changed by resetting its *x* and *y* properties. The **`sbHorizontal`** and **`sbVertical`** are placed on the right and at the bottom of the border pane (lines 26 and 27), respectively.

You can specify the properties of the scroll bar. By default, the property value is **`100`** for **`max`**, **`0`** for **`min`**, **`10`** for **`blockIncrement`**, and **`15`** for **`visibleAmount`**.

A listener is registered to listen for the **`sbHorizontal value`** property change (lines 30–32). When the value of the scroll bar changes, the listener is notified by invoking the handler to set a new *x* value for the text that corresponds to the current value of **`sbHorizontal`** (lines 31 and 32).

A listener is registered to listen for the **`sbVertical value`** property change (lines 35–37). When the value of the scroll bar changes, the listener is notified by invoking the handler to set a new *y* value for the text that corresponds to the current value of **`sbVertical`** (lines 36 and 37).

Alternatively, the code in lines 30–37 can be replaced by using binding properties as follows:

```
text.xProperty().bind(sbHorizontal.valueProperty().
  multiply(paneForText.widthProperty()).
  divide(sbHorizontal.maxProperty()));

text.yProperty().bind(sbVertical.valueProperty().multiply(
  paneForText.heightProperty().divide(
  sbVertical.maxProperty())));
```

Check Point

16.10.1 How do you create a horizontal scroll bar? How do you create a vertical scroll bar?

16.10.2 How do you write the code to respond to the **`value`** property change of a scroll bar?

16.10.3 How do you get the value from a scroll bar? How do you get the maximum value from a scroll bar?

16.11 Slider

Key Point

`Slider` is similar to `ScrollBar`, but `Slider` has more properties and can appear in many forms.

VideoNote
Use `Slider`

Figure 16.24 shows two sliders. `Slider` lets the user graphically select a value by sliding a knob within a bounded interval. The slider can show both major and minor tick marks between them. The number of pixels between the tick marks is specified by the `majorTickUnit` and `minorTickUnit` properties. Sliders can be displayed horizontally or vertically, with or without ticks, and with or without labels.

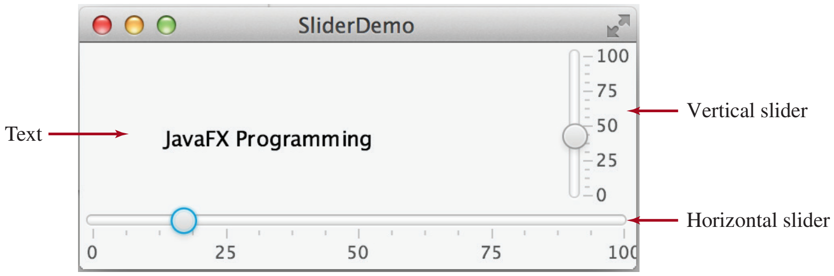
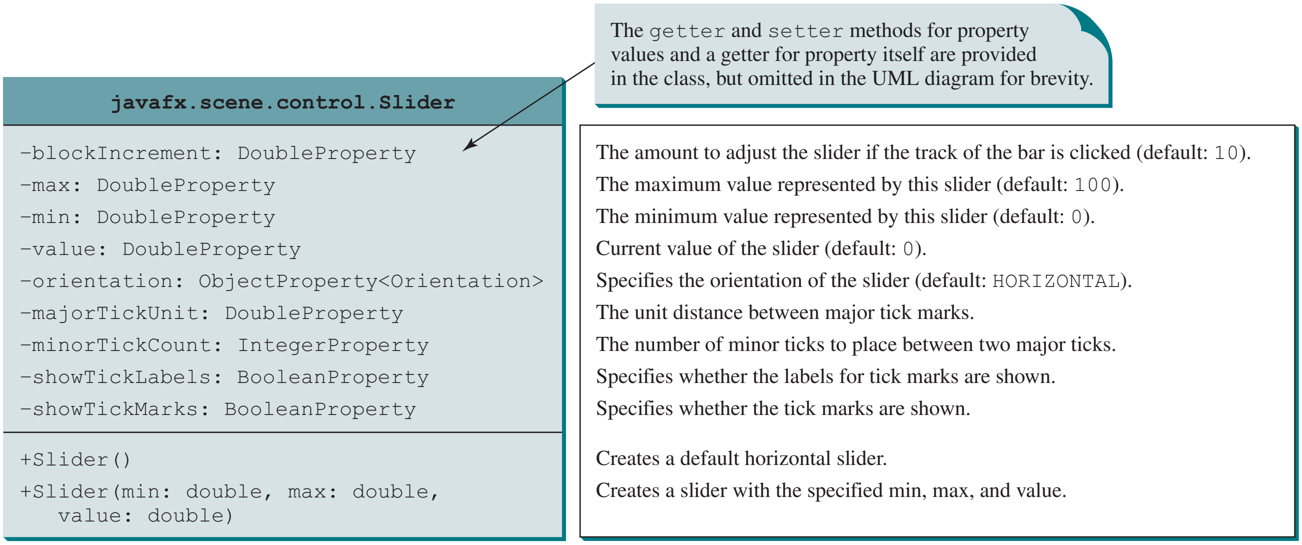


FIGURE 16.24 The sliders move the message on a pane horizontally and vertically. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

The frequently used constructors and properties in `Slider` are shown in Figure 16.25.



The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

`javafx.scene.control.Slider`	
`-blockIncrement: DoubleProperty`	The amount to adjust the slider if the track of the bar is clicked (default: `10`).
`-max: DoubleProperty`	The maximum value represented by this slider (default: `100`).
`-min: DoubleProperty`	The minimum value represented by this slider (default: `0`).
`-value: DoubleProperty`	Current value of the slider (default: `0`).
`-orientation: ObjectProperty<Orientation>`	Specifies the orientation of the slider (default: `HORIZONTAL`).
`-majorTickUnit: DoubleProperty`	The unit distance between major tick marks.
`-minorTickCount: IntegerProperty`	The number of minor ticks to place between two major ticks.
`-showTickLabels: BooleanProperty`	Specifies whether the labels for tick marks are shown.
`-showTickMarks: BooleanProperty`	Specifies whether the tick marks are shown.
`+Slider()`	Creates a default horizontal slider.
`+Slider(min: double, max: double, value: double)`	Creates a slider with the specified min, max, and value.

FIGURE 16.25 `Slider` enables the user to select from a range of values.

Note
The values of a vertical scroll bar increase from top to bottom, but the values of a vertical slider decrease from top to bottom.

You can add a listener to listen for the `value` property change in a slider in the same way as in a scroll bar. We now rewrite the program in the preceding section using the sliders to move a text displayed on a pane in Listing 16.11. A sample run of the program is shown in Figure 16.24.

LISTING 16.11 SliderDemo.java

```
 1  import javafx.application.Application;
 2  import javafx.stage.Stage;
 3  import javafx.geometry.Orientation;
 4  import javafx.scene.Scene;
 5  import javafx.scene.control.Slider;
 6  import javafx.scene.layout.BorderPane;
 7  import javafx.scene.layout.Pane;
 8  import javafx.scene.text.Text;
 9
10  public class SliderDemo extends Application {
11    @Override // Override the start method in the Application class
12    public void start(Stage primaryStage) {
13      Text text = new Text(20, 20, "JavaFX Programming");
14
15      Slider slHorizontal = new Slider();
16      slHorizontal.setShowTickLabels(true);
17      slHorizontal.setShowTickMarks(true);
18
19      Slider slVertical = new Slider();
20      slVertical.setOrientation(Orientation.VERTICAL);
21      slVertical.setShowTickLabels(true);
22      slVertical.setShowTickMarks(true);
23      slVertical.setValue(100);
24
25      // Create a text in a pane
26      Pane paneForText = new Pane();
27      paneForText.getChildren().add(text);
28
29      // Create a border pane to hold text and scroll bars
30      BorderPane pane = new BorderPane();
31      pane.setCenter(paneForText);
32      pane.setBottom(slHorizontal);
33      pane.setRight(slVertical);
34
35      slHorizontal.valueProperty().addListener(ov ->
36        text.setX(slHorizontal.getValue() * paneForText.getWidth() /
37          slHorizontal.getMax()));
38
39      slVertical.valueProperty().addListener(ov ->
40        text.setY((slVertical.getMax() – slVertical.getValue())
41          * paneForText.getHeight() / slVertical.getMax()));
42
43      // Create a scene and place it in the stage
44      Scene scene = new Scene(pane, 450, 170);
45      primaryStage.setTitle("SliderDemo"); // Set the stage title
46      primaryStage.setScene(scene); // Place the scene in the stage
47      primaryStage.show(); // Display the stage
48    }
49  }
```

horizontal slider
set slider properties
vertical slider
set slider properties
add text to a pane
border pane
set new location for text
set new location for text

`Slider` is similar to **`ScrollBar`** but has more features. As shown in this example, you can specify labels, major ticks, and minor ticks on a **`Slider`** (lines 16 and 17).

A listener is registered to listen for the **`slHorizontal value`** property change (lines 35–37) and another one is for the **`sbVertical value`** property change (lines 39–41). When the value of the slider changes, the listener is notified by invoking the handler to set a new position for the text (lines 36 and 37 and 40 and 41). Note since the value of a vertical slider decreases from top to bottom, the corresponding *y* value for the text is adjusted accordingly.

The code in lines 35–41 can be replaced by using binding properties as follows:

```
text.xProperty().bind(slHorizontal.valueProperty().
  multiply(paneForText.widthProperty()).
  divide(slHorizontal.maxProperty()));

text.yProperty().bind((slVertical.maxProperty().subtract(
  slVertical.valueProperty()).multiply(
  paneForText.heightProperty().divide(
  slVertical.maxProperty()))));
```

Listing 15.17 gives a program that displays a bouncing ball. You can add a slider to control the speed of the ball movement, as shown in Figure 16.26. The new program is given in Listing 16.12.

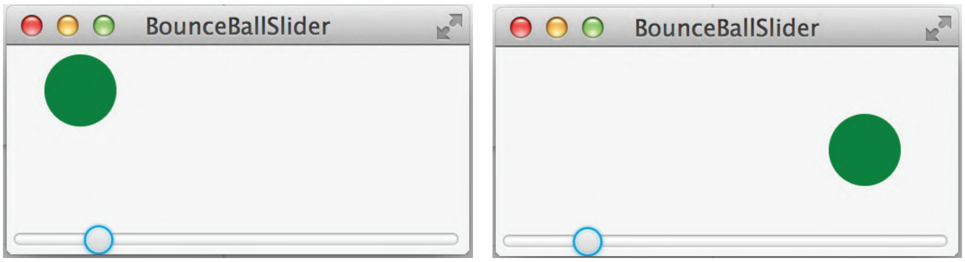


FIGURE 16.26 You can increase or decrease the speed of the ball using a slider. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

LISTING 16.12 BounceBallSlider.java

create a ball pane
create a slider
set max value for slider
bind rate with slider value

create a border pane
add ball pane to center
add slider to the bottom

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Slider;
import javafx.scene.layout.BorderPane;

public class BounceBallSlider extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    BallPane ballPane = new BallPane();
    Slider slSpeed = new Slider();
    slSpeed.setMax(20);
    ballPane.rateProperty().bind(slSpeed.valueProperty());

    BorderPane pane = new BorderPane();
    pane.setCenter(ballPane);
    pane.setBottom(slSpeed);

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 250, 250);
    primaryStage.setTitle("BounceBallSlider"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

The `BallPane` class defined in Listing 15.17 animates a ball bouncing in a pane. The `rateProperty()` method in `BallPane` returns a property value for the animation rate. The animation stops if the rate is 0. If the rate is greater than 20, the animation will be too fast. Therefore, we purposely set the rate to a value between 0 and 20. This value is bound to the slider value (line 13). Thus, the slider max value is set to 20 (line 12).

Check Point

16.11.1 How do you create a horizontal slider? How do you create a vertical slider?

16.11.2 How do you add a listener to handle the property value change of a slider?

16.11.3 How do you get the value from a slider? How do you get the maximum value from a slider?

16.12 Case Study: Developing a Tic-Tac-Toe Game

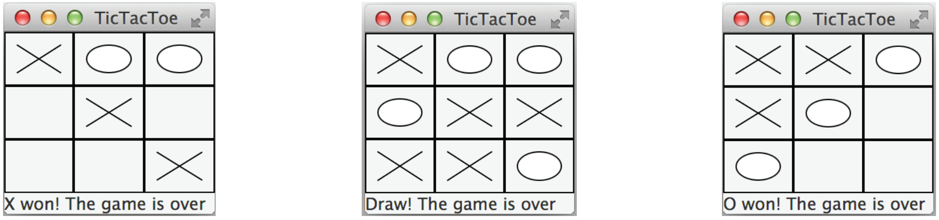
Key Point

This section develops a program for playing tic-tac-toe game.

From the many examples in this and earlier chapters, you have learned about objects, classes, arrays, class inheritance, GUI, and event-driven programming. Now it is time to put what you have learned to work in developing comprehensive projects. In this section, we will develop a JavaFX program with which to play the popular game of tic-tac-toe.

VideoNote
Tic-Tac-Toe

Two players take turns marking an available cell in a 3 × 3 grid with their respective tokens (either X or O). When one player has placed three tokens in a horizontal, vertical, or diagonal row on the grid, the game is over and that player has won. A draw (no winner) occurs when all the cells on the grid have been filled with tokens and neither player has achieved a win. Figure 16.27 shows the representative sample runs of the game.



X won! The game is over

Draw! The game is over

O won! The game is over

(a) The X player won the game (b) Draw—no winners (c) The O player won the game

Figure 16.27 Two players play a tic-tac-toe game. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

All the examples you have seen so far show simple behaviors that are easy to model with classes. The behavior of the tic-tac-toe game is somewhat more complex. To define classes that model the behavior, you need to study and understand the game.

Assume all the cells are initially empty, and that the first player takes the X token and the second player the O token. To mark a cell, the player points the mouse to the cell and clicks it. If the cell is empty, the token (X or O) is displayed. If the cell is already filled, the player's action is ignored.

From the preceding description, it should be obvious that a cell is a GUI object that handles the mouse-click event and displays tokens. There are many choices for this object. We will use a pane to model a cell and to display a token (X or O). How do you know the state of the cell (empty, X, or O)? You use a property named `token` of the `char` type in the `Cell` class. The `Cell` class is responsible for drawing the token when an empty cell is clicked, so you need to write the code for listening to the mouse-clicked action and for painting the shapes for tokens X and O. The `Cell` class can be defined as shown in Figure 16.28.

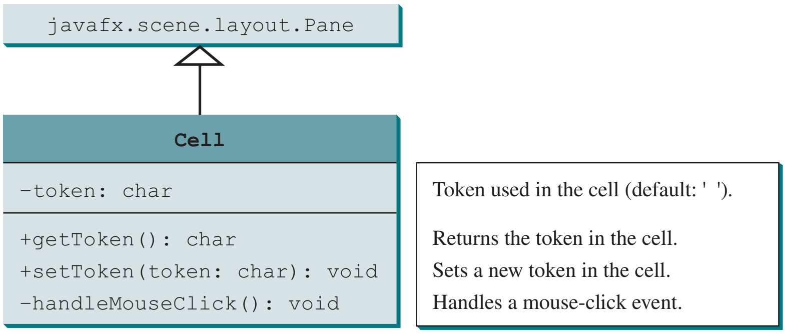

javafx.scene.layout.Pane
Cell
-token: char
+getToken(): char
+setToken(token: char): void
-handleMouseClick(): void
Token used in the cell (default: ' ').
Returns the token in the cell.
Sets a new token in the cell.
Handles a mouse-click event.

FIGURE 16.28 The `Cell` class displays the token in a cell.

The tic-tac-toe board consists of nine cells, created using `new Cell[3][3]`. To determine which player's turn it is, you can introduce a variable named `whoseTurn` of the `char` type. `whoseTurn` is initially `'X'`, then changes to `'O'`, and subsequently changes between `'X'` and `'O'` whenever a new cell is occupied. When the game is over, set `whoseTurn` to `' '`.

How do you know whether the game is over, whether there is a winner, and who is the winner, if any? You can define a method named `isWon(char token)` to check whether a specified token has won and a method named `isFull()` to check whether all the cells are occupied.

Clearly, two classes emerge from the foregoing analysis. One is the `Cell` class, which handles operations for a single cell; the other is the `TicTacToe` class, which plays the whole game and deals with all the cells. The relationship between these two classes is shown in Figure 16.29.

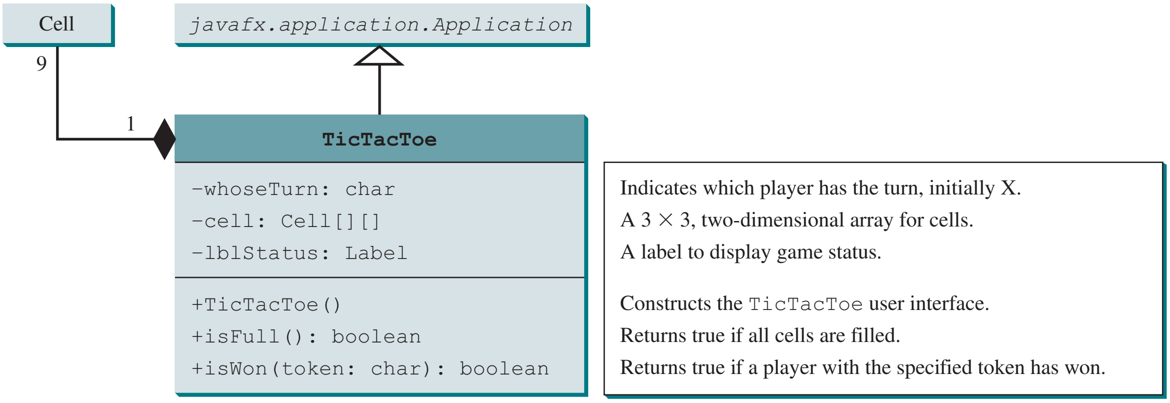

Cell
9
1
javafx.application.Application
TicTacToe
-whoseTurn: char
-cell: Cell[][]
-lblStatus: Label
+TicTacToe()
+isFull(): boolean
+isWon(token: char): boolean
Indicates which player has the turn, initially X.
A 3 × 3, two-dimensional array for cells.
A label to display game status.
Constructs the TicTacToe user interface.
Returns true if all cells are filled.
Returns true if a player with the specified token has won.

FIGURE 16.29 The `TicTacToe` class contains nine cells.

Since the `Cell` class is only to support the `TicTacToe` class, it can be defined as an inner class in `TicTacToe`. The complete program is given in Listing 16.13.

LISTING 16.13 TicTacToe.java

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.scene.shape.Ellipse;

public class TicTacToe extends Application {
  // Indicate which player has a turn, initially it is the X player
  private char whoseTurn = 'X';

  // Create and initialize cell
  private Cell[][] cell = new Cell[3][3];

  // Create and initialize a status label
  private Label lblStatus = new Label("X's turn to play");

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Pane to hold cell
    GridPane pane = new GridPane();
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        pane.add(cell[i][j] = new Cell(), j, i);

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(pane);
    borderPane.setBottom(lblStatus);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 450, 170);
    primaryStage.setTitle("TicTacToe"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /** Determine if the cell are all occupied */
  public boolean isFull() {
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        if (cell[i][j].getToken() == ' ')
          return false;

    return true;
  }

  /** Determine if the player with the specified token wins */
  public boolean isWon(char token) {
    for (int i = 0; i < 3; i++)
      if (cell[i][0].getToken() == token
          && cell[i][1].getToken() == token
          && cell[i][2].getToken() == token) {
        return true;
      }

    for (int j = 0; j < 3; j++)
      if (cell[0][j].getToken() == token
          && cell[1][j].getToken() == token
          && cell[2][j].getToken() == token) {
        return true;
      }
```

main class `TicTacToe`

hold nine cells

create a cell

tic-tac-toe cells in center

label at bottom

check `isFull`

check rows

check columns

check major diagonal

```

    if (cell[0][0].getToken() == token
        && cell[1][1].getToken() == token
        && cell[2][2].getToken() == token) {
      return true;
    }

```

check subdiagonal

```
    if (cell[0][2].getToken() == token
        && cell[1][1].getToken() == token
        && cell[2][0].getToken() == token) {
      return true;
    }

    return false;
  }

  // An inner class for a cell
```

inner class Cell

```
  public class Cell extends Pane {
    // Token used for this cell
    private char token = ' ';

    public Cell() {
      setStyle("-fx-border-color: black");
      this.setPrefSize(2000, 2000);
```

register listener

```
      this.setOnMouseClicked(e -> handleMouseClick());
    }

    /** Return token */
    public char getToken() {
      return token;
    }

    /** Set a new token */
    public void setToken(char c) {
      token = c;

```

display X

```
      if (token == 'X') {
        Line line1 = new Line(10, 10,
          this.getWidth() - 10, this.getHeight() - 10);
        line1.endXProperty().bind(this.widthProperty().subtract(10));
        line1.endYProperty().bind(this.heightProperty().subtract(10));
        Line line2 = new Line(10, this.getHeight() - 10,
          this.getWidth() - 10, 10);
        line2.startYProperty().bind(
          this.heightProperty().subtract(10));
        line2.endXProperty().bind(this.widthProperty().subtract(10));

        // Add the lines to the pane
        this.getChildren().addAll(line1, line2);
      }
```

display O

```
      else if (token == 'O') {
        Ellipse ellipse = new Ellipse(this.getWidth() / 2,
          this.getHeight() / 2, this.getWidth() / 2 - 10,
          this.getHeight() / 2 - 10);
        ellipse.centerXProperty().bind(
          this.widthProperty().divide(2));
        ellipse.centerYProperty().bind(
          this.heightProperty().divide(2));
        ellipse.radiusXProperty().bind(
          this.widthProperty().divide(2).subtract(10));
```

```
126           ellipse.radiusYProperty().bind(
127             this.heightProperty().divide(2).subtract(10));
128           ellipse.setStroke(Color.BLACK);
129           ellipse.setFill(Color.WHITE);
130
131           getChildren().add(ellipse); // Add the ellipse to the pane
132         }
133       }
134
135       /* Handle a mouse click event */
136       private void handleMouseClick() {
137         // If cell is empty and game is not over
138         if (token == ' ' && whoseTurn != ' ') {
139           setToken(whoseTurn); // Set token in the cell
140
141           // Check game status
142           if (isWon(whoseTurn)) {
143             lblStatus.setText(whoseTurn + " won! The game is over");
144             whoseTurn = ' '; // Game is over
145           }
146           else if (isFull()) {
147             lblStatus.setText("Draw! The game is over");
148             whoseTurn = ' '; // Game is over
149           }
150           else {
151             // Change the turn
152             whoseTurn = (whoseTurn == 'X') ? 'O' : 'X';
153             // Display whose turn
154             lblStatus.setText(whoseTurn + "'s turn");
155           }
156         }
157       }
158     }
159   }
```

handle mouse click

The **`TicTacToe`** class initializes the user interface with nine cells placed in a grid pane (lines 25–28). A label named **`lblStatus`** is used to show the status of the game (line 20). The variable **`whoseTurn`** (line 14) is used to track the next type of token to be placed in a cell. The methods **`isFull`** (lines 42–49) and **`isWon`** (lines 52–80) are for checking the status of the game.

Since **`Cell`** is an inner class in **`TicTacToe`**, the variable **`whoseTurn`** and methods **`isFull`** and **`isWon`** defined in **`TicTacToe`** can be referenced from the **`Cell`** class. The inner class makes programs simple and concise. If **`Cell`** were not defined as an inner class of **`TicTacToe`**, you would have to pass an object of **`TicTacToe`** to **`Cell`** in order for the variables and methods in **`TicTacToe`** to be used in **`Cell`**.

The listener for the mouse-click action is registered for the cell (line 90). If an empty cell is clicked and the game is not over, a token is set in the cell (line 138). If the game is over, **`whoseTurn`** is set to **`' '`** (lines 144 and 148). Otherwise, **`whoseTurn`** is alternated to a new turn (line 152).

Tip

Use an incremental approach in developing and testing a Java project of this kind. For example, this program can be divided into five steps:

incremental development and testing

1. Lay out the user interface and display a fixed token X on a cell.
2. Enable the cell to display a fixed token X upon a mouse click.
3. Coordinate between the two players so as to display tokens X and O alternately.
4. Check whether a player wins, or whether all the cells are occupied without a winner.
5. Implement displaying a message on the label upon each move by a player.

Check Point

16.12.1 When the game starts, what value is in `whoseTurn`? When the game is over, what value is in `whoseTurn`?

16.12.2 What happens when the user clicks on an empty cell if the game is not over? What happens when the user clicks on an empty cell if the game is over?

16.12.3 How does the program check whether a player wins? How does the program check whether all cells are filled?

16.13 Video and Audio

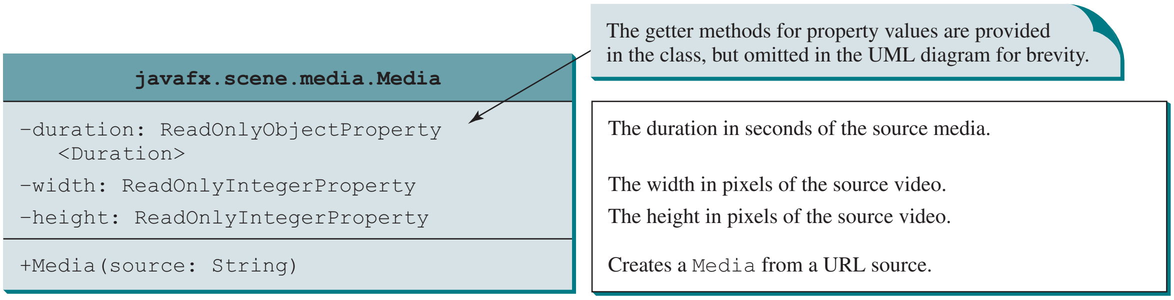
Key Point

You can use the `Media` class to obtain the source of the media, the `MediaPlayer` class to play and control the media, and the `MediaView` class to display the video.

VideoNote
Use Media, MediaPlayer, and MediaView

Media (video and audio) is essential in developing rich GUI applications. JavaFX provides the `Media`, `MediaPlayer`, and `MediaView` classes for working with media. Currently, JavaFX supports MP3, AIFF, WAV, and MPEG-4 audio formats and FLV and MPEG-4 video formats.

The `Media` class represents a media source with properties `duration`, `width`, and `height`, as shown in Figure 16.30. You can construct a `Media` object from an Internet URL string.

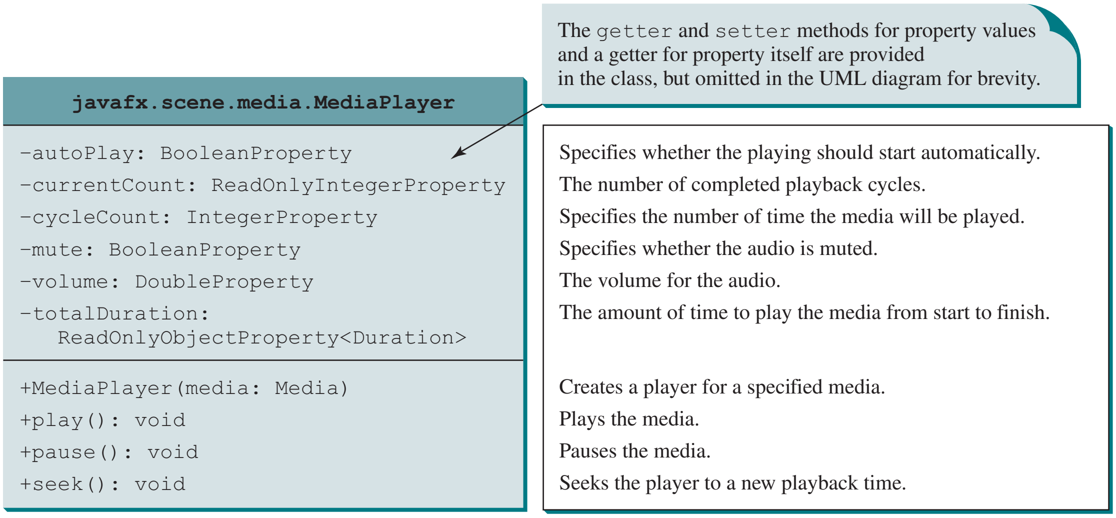


The getter methods for property values are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.media.Media	
-duration: ReadOnlyObjectProperty <Duration>	The duration in seconds of the source media.
-width: ReadOnlyIntegerProperty	The width in pixels of the source video.
-height: ReadOnlyIntegerProperty	The height in pixels of the source video.
+Media(source: String)	Creates a Media from a URL source.

FIGURE 16.30 `Media` represents a media source such as a video or an audio.

The `MediaPlayer` class plays and controls the media with properties such as `autoPlay`, `currentCount`, `cycleCount`, `mute`, `volume`, and `totalDuration`, as shown in Figure 16.31. You can construct a `MediaPlayer` object from a media and use the `pause()` and `play()` methods to pause and resume playing.



The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.media.MediaPlayer	
-autoPlay: BooleanProperty	Specifies whether the playing should start automatically.
-currentCount: ReadOnlyIntegerProperty	The number of completed playback cycles.
-cycleCount: IntegerProperty	Specifies the number of time the media will be played.
-mute: BooleanProperty	Specifies whether the audio is muted.
-volume: DoubleProperty	The volume for the audio.
-totalDuration: ReadOnlyObjectProperty<Duration>	The amount of time to play the media from start to finish.
+MediaPlayer(media: Media)	Creates a player for a specified media.
+play(): void	Plays the media.
+pause(): void	Pauses the media.
+seek(): void	Seeks the player to a new playback time.

FIGURE 16.31 `MediaPlayer` plays and controls a media.

The **`MediaView`** class is a subclass of **`Node`** that provides a view of the **`Media`** being played by a **`MediaPlayer`**. The **`MediaView`** class provides the properties for viewing the media, as shown in Figure 16.32.

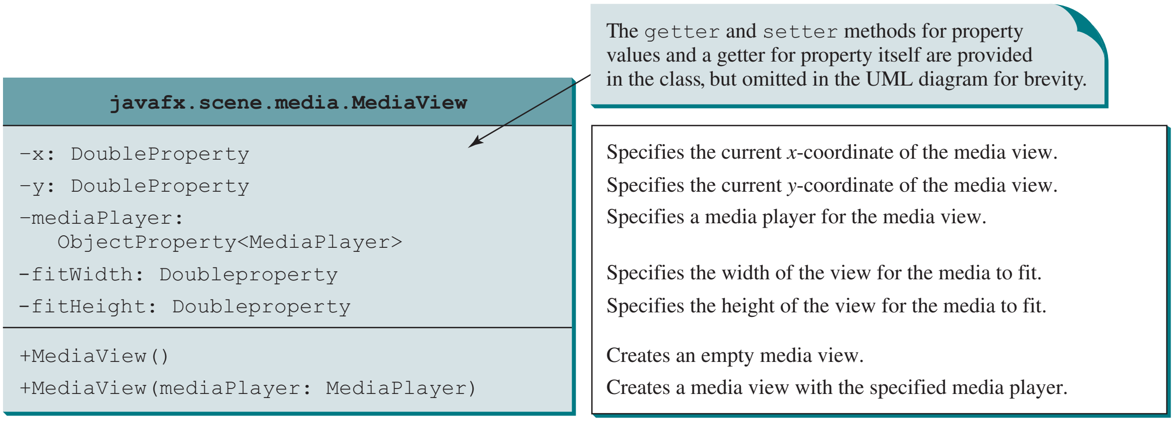

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
javafx.scene.media.MediaView
-x: DoubleProperty
-y: DoubleProperty
-mediaPlayer: ObjectProperty<MediaPlayer>
-fitWidth: Doubleproperty
-fitHeight: Doubleproperty
+MediaView()
+MediaView(mediaPlayer: MediaPlayer)
Specifies the current *x*-coordinate of the media view.
Specifies the current *y*-coordinate of the media view.
Specifies a media player for the media view.
Specifies the width of the view for the media to fit.
Specifies the height of the view for the media to fit.
Creates an empty media view.
Creates a media view with the specified media player.

FIGURE 16.32 **`MediaView`** provides the properties for viewing the media.

Listing 16.14 gives an example that displays a video in a view, as shown in Figure 16.33. You can use the play/pause button to play or pause the video and use the rewind button to restart the video, and use the slider to control the volume of the audio.

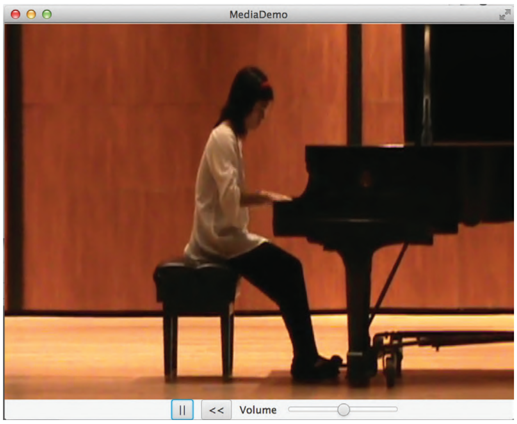

MediaDemo
||
<<
Volume

FIGURE 16.33 The program controls and plays a video.

LISTING 16.14 `MediaDemo.java`

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Region;
import javafx.scene.media.Media;
```

create a media
create a media player
create a media view
create a play/pause button
add handler for button action
play media
pause media
create a rewind button
create a handler for rewinding
create a slider for volume
set current volume
bind volume with slider
add buttons, slider to hBox
place media view in a pane

```
12 import javafx.scene.media.MediaPlayer;
13 import javafx.scene.media.MediaView;
14 import javafx.util.Duration;
15
16 public class MediaDemo extends Application {
17   private static final String MEDIA_URL =
18     "http://liveexample.pearsoncmg.com/common/sample.mp4";
19
20   @Override // Override the start method in the Application class
21   public void start(Stage primaryStage) {
22     Media media = new Media(MEDIA_URL);
23     MediaPlayer mediaPlayer = new MediaPlayer(media);
24     MediaView mediaView = new MediaView(mediaPlayer);
25
26     Button playButton = new Button(">");
27     playButton.setOnAction(e -> {
28       if (playButton.getText().equals(">")) {
29         mediaPlayer.play();
30         playButton.setText("||");
31       } else {
32         mediaPlayer.pause();
33         playButton.setText(">");
34       }
35     });
36
37     Button rewindButton = new Button("<<");
38     rewindButton.setOnAction(e -> mediaPlayer.seek(Duration.ZERO));
39
40     Slider slVolume = new Slider();
41     slVolume.setPrefWidth(150);
42     slVolume.setMaxWidth(Region.USE_PREF_SIZE);
43     slVolume.setMinWidth(30);
44     slVolume.setValue(50);
45     mediaPlayer.volumeProperty().bind(
46       slVolume.valueProperty().divide(100));
47
48     HBox hBox = new HBox(10);
49     hBox.setAlignment(Pos.CENTER);
50     hBox.getChildren().addAll(playButton, rewindButton,
51       new Label("Volume"), slVolume);
52
53     BorderPane pane = new BorderPane();
54     pane.setCenter(mediaView);
55     pane.setBottom(hBox);
56
57     // Create a scene and place it in the stage
58     Scene scene = new Scene(pane, 650, 500);
59     primaryStage.setTitle("MediaDemo"); // Set the stage title
60     primaryStage.setScene(scene); // Place the scene in the stage
61     primaryStage.show(); // Display the stage
62   }
63 }
```

The source of the media is a URL string defined in lines 17 and 18. The program creates a `Media` object from this URL (line 22), a `MediaPlayer` from the `Media` object (line 23), and a `MediaView` from the `MediaPlayer` object (line 24). The relationship among these three objects is shown in Figure 16.34.

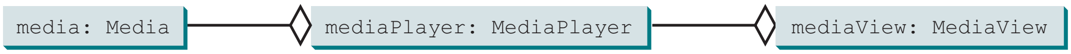


FIGURE 16.34 The media represents the source, the media player controls the playing, and the media view displays the video.

A `Media` object supports live streaming. You can now download a large media file and play it in the same time. A `Media` object can be shared by multiple media players and different views can use the same `MediaPlayer` object.

A play button is created (line 26) to play/pause the media (line 29). The button's text is changed to `||` (line 30) if the button's current text is `>` (line 28). If the button's current text is `||`, it is changed to `>` (line 33) and the player is paused (line 32).

A rewind button is created (line 37) to reset the playback time to the beginning of the media stream by invoking `seek(Duration.ZERO)` (line 38).

A slider is created (line 40) to set the volume. The media player's volume property is bound to the slider (lines 45 and 46).

The buttons and slider are placed in an `HBox` (lines 48–51) and the media view is placed in the center of the border pane (line 54) and the `HBox` is placed at the bottom of the border pane (line 55).

Check Point

16.13.1 How do you create a `Media` from a URL? How do you create a `MediaPlayer`? How do you create a `MediaView`?

16.13.2 If the URL is typed as liveexample.pearsoncmg.com/common/sample.mp4 without http:// in front of it, will it work?

16.13.3 Can you place a `Media` in multiple `MediaPlayer`s? Can you place a `MediaPlayer` in multiple `MediaView`s? Can you place a `MediaView` in multiple `Pane`s?

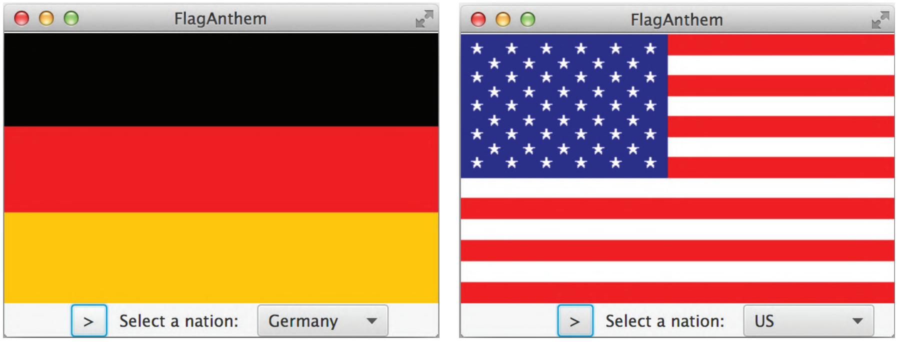
16.14 Case Study: National Flags and Anthems

Key Point

This case study presents a program that displays a nation's flag and plays its anthem.

The images for seven national flags, named **flag0.gif**, **flag1.gif**, . . . , **flag6.gif** for Denmark, Germany, China, India, Norway, the United Kingdom, and the United States are stored under http://liveexample.pearsoncmg.com/common/image. The audio consists of national anthems for these seven nations, named **anthem0.mp3**, **anthem1.mp3**, . . . , **anthem6.mp3**. They are stored under http://liveexample.pearsoncmg.com/common/audio.

The program enables the user to select a nation from a combo box, then displays its flag and plays its anthem. The user can suspend the audio by clicking the `||` button, and resume it by clicking the `<` button, as shown in Figure 16.35.



Select a nation: Germany

Select a nation: US

FIGURE 16.35 The program displays a national flag and plays its anthem. *Source*: booka/Fotolia.

The program is given in Listing 16.15.

LISTING 16.15 FlagAnthem.java

URLBase for image and audio
track current image/audio
image array
media player array
load image
load audio
create play button
handle button action
play audio
pause audio
create image view
create combo box
create observable list
process combo selection
choose a new nation
play audio

```java
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.stage.Stage;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;

public class FlagAnthem extends Application {
  private final static int NUMBER_OF_NATIONS = 7;
  private final static String URLBase =
    "https://liveexample.pearsoncmg.com/common";
  private int currentIndex = 0;

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    Image[] images = new Image[NUMBER_OF_NATIONS];
    MediaPlayer[] mp = new MediaPlayer[NUMBER_OF_NATIONS];

    // Load images and audio
    for (int i = 0; i < NUMBER_OF_NATIONS; i++) {
      images[i] = new Image(URLBase + "/image/flag" + i + ".gif");
      mp[i] = new MediaPlayer(new Media(
        URLBase + "/audio/anthem/anthem" + i + ".mp3"));
    }

    Button btPlayPause = new Button("||");
    btPlayPause.setOnAction(e -> {
      if (btPlayPause.getText().equals(">")) {
        btPlayPause.setText("||");
        mp[currentIndex].play();
      }
      else {
        btPlayPause.setText(">");
        mp[currentIndex].pause();
      }
    });

    ImageView imageView = new ImageView(images[currentIndex]);
    ComboBox<String> cboNation = new ComboBox<>();
    ObservableList<String> items = FXCollections.observableArrayList
      ("Denmark", "Germany", "China", "India", "Norway", "UK", "US");
    cboNation.getItems().addAll(items);
    cboNation.setValue(items.get(0));
    cboNation.setOnAction(e -> {
      mp[currentIndex].stop();
      currentIndex = items.indexOf(cboNation.getValue());
      imageView.setImage(images[currentIndex]);
      mp[currentIndex].play();
58          btPlayPause.setText("||");
59        });
60
61        HBox hBox = new HBox(10);
62        hBox.getChildren().addAll(btPlayPause,
63          new Label("Select a nation: "), cboNation);
64        hBox.setAlignment(Pos.CENTER);
65
66        // Create a pane to hold nodes
67        BorderPane pane = new BorderPane();
68        pane.setCenter(imageView);
69        pane.setBottom(hBox);
70
71        // Create a scene and place it in the stage
72        Scene scene = new Scene(pane, 350, 270);
73        primaryStage.setTitle("FlagAnthem"); // Set the stage title
74        primaryStage.setScene(scene); // Place the scene in the stage
75        primaryStage.show(); // Display the stage
76        mp[currentIndex].play(); // Play the current selected anthem
77      }
78    }
```

The program loads the image and audio from the Internet (lines 29–33). A play/pause button is created to control the playing of the audio (line 35). When the button is clicked, if the button's current text is `>` (line 37), its text is changed to `||` (line 38) and the player is paused (line 39); If the button's current text is `||`, it is changed to `>` (line 42) and the player is paused (line 43).

An image view is created to display a flag image (line 47). A combo box is created for selecting a nation (line 48–51). When a new country name in the combo box is selected, the current audio is stopped (line 54), the newly selected nation's image is displayed (line 56) and the new anthem is played (line 57).

JavaFX also provides the `AudioClip` class for creating auto clips. An `AudioClip` object can be created using `new AudioClip(URL)`. An audio clip stores the audio in memory. `AudioClip` is more efficient for playing a small audio clip in the program than using `MediaPlayer`. `AudioClip` has the similar methods as in the `MediaPlayer` class.

16.14.1 In Listing 16.15, which code sets the initial image icon and which code plays the audio?

16.14.2 In Listing 16.15, what does the program do when a new nation is selected in the combo box?

Check Point

CHAPTER SUMMARY

1. The abstract `Labeled` class is the base class for `Label`, `Button`, `CheckBox`, and `RadioButton`. It defines properties `alignment`, `contentDisplay`, `text`, `graphic`, `graphicTextGap`, `textFill`, `underline`, and `wrapText`.

2. The abstract `ButtonBase` class is the base class for `Button`, `CheckBox`, and `RadioButton`. It defines the `onAction` property for specifying a handler for action events.

3. The abstract `TextInputContorl` class is the base class for `TextField` and `TextArea`. It defines the properties `text` and `editable`.

4. A `TextField` fires an action event when clicking the *Enter* key with the text field focused. A `TextArea` is often used for editing a multiline text.

5. `ComboBox<T>` and `ListView<T>` are generic classes for storing elements of type `T`. The elements in a combo box or a list view are stored in an observable list.

6. A `ComboBox` fires an action event when a new item is selected.

7. You can set a single item or multiple items selection for a `ListView` and add a listener for processing selected items.

8. You can use a `ScrollBar` or `Slider` to select a range of values and add a listener to the `value` property to respond to the change of the value.

9. JavaFX provides the `Media` class for loading a media, the `MediaPlayer` class for controlling a media, and the `MediaView` for displaying a media.

QUIZ

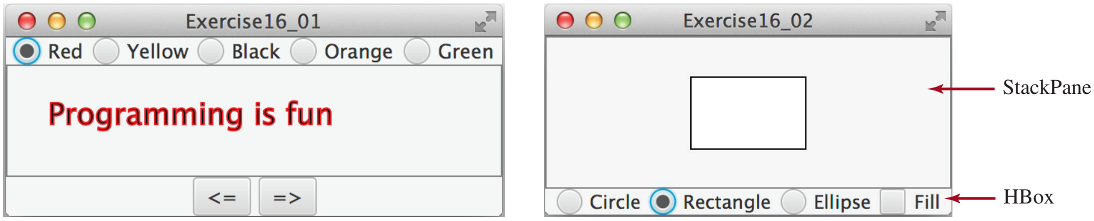
Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

Sections 16.2–16.5

***16.1** *(Use radio buttons)* Write a GUI program as shown in Figure 16.36a. You can use buttons to move the message to the left and right and use the radio buttons to change the color for the message displayed.

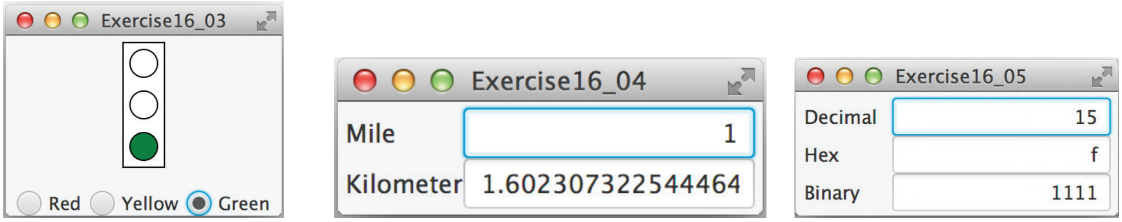


(a) (b)

FIGURE 16.36 (a) The <= and => buttons move the message, and the radio buttons change the color for the message. (b) The program displays a circle, rectangle, and ellipse when you select a shape type. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***16.2** *(Select geometric figures)* Write a program that draws various figures, as shown in Figure 16.36b. The user selects a figure from a radio button and uses a check box to specify whether it is filled.

****16.3** *(Traffic lights)* Write a program that simulates a traffic light. The program lets the user select one of three lights: red, yellow, or green. When a radio button is selected, the light is turned on. Only one light can be on at a time (see Figure 16.37a). No light is on when the program starts.



Exercise16_03

○ Red ○ Yellow ● Green

Exercise16_04

Mile 1

Kilometer 1.602307322544464

Exercise16_05

Decimal 15

Hex f

Binary 1111

(a) (b) (c)

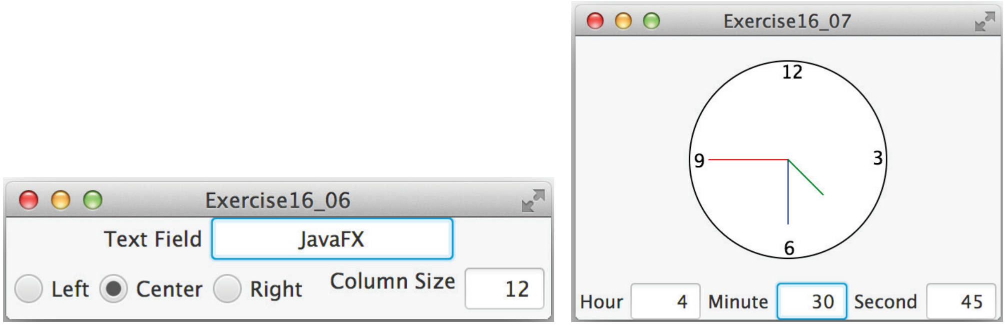
FIGURE 16.37 (a) The radio buttons are grouped to let you turn only one light on at a time. (b) The program converts miles to kilometers and vice versa. (c) The program converts among decimal, hex, and binary numbers. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***16.4** *(Create a miles/kilometers converter)* Write a program that converts miles and kilometers, as shown in Figure 16.37b. If you enter a value in the Mile text field and press the *Enter* key, the corresponding kilometer measurement is displayed in the Kilometer text field. Likewise, if you enter a value in the Kilometer text field and press the *Enter* key, the corresponding miles is displayed in the Mile text field.

***16.5** *(Convert numbers)* Write a program that converts among decimal, hex, and binary numbers, as shown in Figure 16.37c. When you enter a decimal value in the decimal-value text field and press the *Enter* key, its corresponding hex and binary numbers are displayed in the other two text fields. Likewise, you can enter values in the other fields and convert them accordingly. (*Hint*: Use the `Integer.parseInt(s, radix)` method to parse a string to a decimal and use `Integer.toHexString(decimal)` and `Integer.toBinaryString(decimal)` to obtain a hex number or a binary number from a decimal.)

***16.6** *(Demonstrate `TextField` properties)* Write a program that sets the horizontal-alignment and column-size properties of a text field dynamically, as shown in Figure 16.38a.

VideoNote
Use radio buttons and text fields



Exercise16_06

Text Field JavaFX

○ Left ● Center ○ Right Column Size 12

Exercise16_07

Hour 4 Minute 30 Second 45

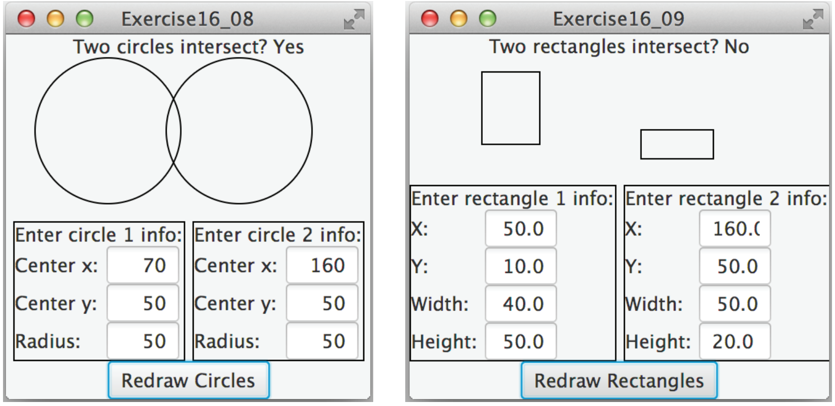
(a) (b)

FIGURE 16.38 (a) You can set a text field's properties for the horizontal alignment and column size dynamically. (b) The program displays the time specified in the text fields. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***16.7** *(Set clock time)* Write a program that displays a clock and sets the time with the input from three text fields, as shown in Figure 16.38b. Use the `ClockPane` in Listing 14.21. Resize the clock to the center of the pane.

****16.8** *(Geometry: two circles intersect?)* Write a program that enables the user to specify the location and size of the circles, and displays whether the two circles intersect, as shown in Figure 16.39a. Enable the user to point the mouse inside a circle and drag it. As the circle is being dragged, the circle's center coordinates in the text fields are updated.

****16.9** *(Geometry: two rectangles intersect?)* Write a program that enables the user to specify the location and size of the rectangles and displays whether the two rectangles intersect, as shown in Figure 16.39b. Enable the user to point the mouse inside a rectangle and drag it. As the rectangle is being dragged, the rectangle's center coordinates in the text fields are updated.



Two circles intersect? Yes

Enter circle 1 info:
Center x: 70
Center y: 50
Radius: 50

Enter circle 2 info:
Center x: 160
Center y: 50
Radius: 50

Two rectangles intersect? No

Enter rectangle 1 info:
X: 50.0
Y: 10.0
Width: 40.0
Height: 50.0

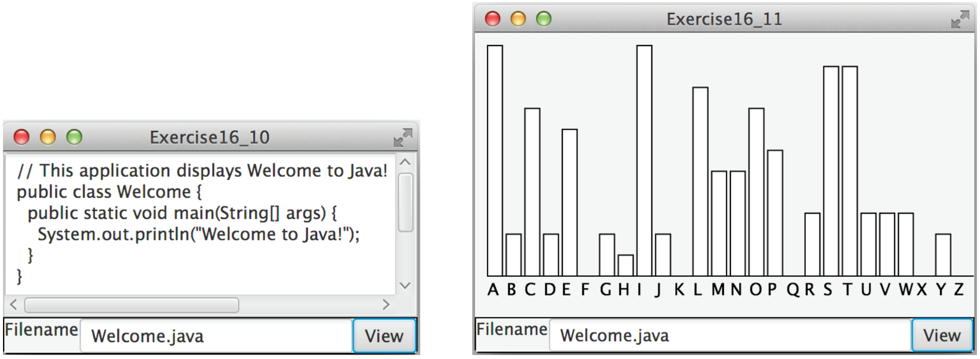
Enter rectangle 2 info:
X: 160.(
Y: 50.0
Width: 50.0
Height: 20.0

(a) (b)

FIGURE 16.39 Check whether two circles and two rectangles are overlapping. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Sections 16.6–16.8

****16.10** *(Text viewer)* Write a program that displays a text file in a text area, as shown in Figure 16.40a. The user enters a file name in a text field and clicks the *View* button; the file is then displayed in a text area.



```
// This application displays Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

Filename Welcome.java

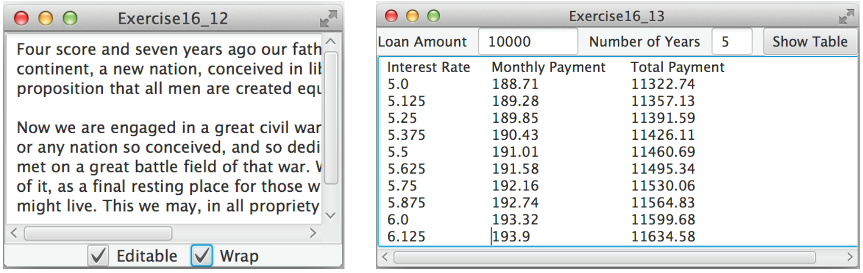
Filename Welcome.java

(a) (b)

FIGURE 16.40 (a) The program displays the text from a file in a text area. (b) The program displays a histogram that shows the occurrences of each letter in the file. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****16.11** *(Create a histogram for occurrences of letters)* Write a program that reads a file and displays a histogram to show the occurrences of each letter in the file, as shown in Figure 16.40b. The file name is entered from a text field. Pressing the *Enter* key on the text field causes the program to start to read, process the file, and display the histogram. The histogram is displayed in the center of the window. Define a class named `Histogram` that extends `Pane`. The class contains the property `counts` that is an array of 26 elements. `counts[0]` stores the number of `A`, `counts[1]` the number of `B`, and so on. The class also contains a setter method for setting a new `counts` and displaying the histogram for the new `counts`.

***16.12** *(Demonstrate `TextArea` properties)* Write a program that demonstrates the properties of a text area. The program uses a check box to indicate whether the text is wrapped onto next line, as shown in Figure 16.41a.



Four score and seven years ago our fath
continent, a new nation, conceived in lil
proposition that all men are created equ

Now we are engaged in a great civil war
or any nation so conceived, and so dedi
met on a great battle field of that war. V
of it, as a final resting place for those w
might live. This we may, in all propriety

☑ Editable ☑ Wrap

Loan Amount 10000 Number of Years 5

```
Interest Rate   Monthly Payment   Total Payment
5.0             188.71            11322.74
5.125           189.28            11357.13
5.25            189.85            11391.59
5.375           190.43            11426.11
5.5             191.01            11460.69
5.625           191.58            11495.34
5.75            192.16            11530.06
5.875           192.74            11564.83
6.0             193.32            11599.68
6.125           193.9             11634.58
```

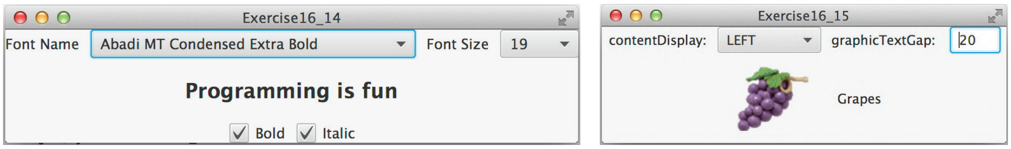
(a) (b)

FIGURE 16.41 (a) You can set the options to enable text editing and text wrapping. (b) The program displays a table for monthly payments and total payments on a given loan based on various interest rates. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***16.13** *(Compare loans with various interest rates)* Rewrite Programming Exercise 5.21 to create a GUI, as shown in Figure 16.41b. Your program should let the user enter the loan amount and loan period in the number of years from text fields, and it should display the monthly and total payments for each interest rate starting from `5`% to `8`%, with increments of one-eighth, in a text area.

****16.14** *(Select a font)* Write a program that can dynamically change the font of a text in a label displayed on a stack pane. The text can be displayed in bold and italic at the same time. You can select the font name or font size from combo boxes, as shown in Figure 16.42a. The available font names can be obtained using `Font.getFontNames()`. The combo box for the font size is initialized with numbers from `1` to `100`.

VideoNote
Set fonts



Font Name: Abadi MT Condensed Extra Bold Font Size: 19

☑ Bold ☑ Italic

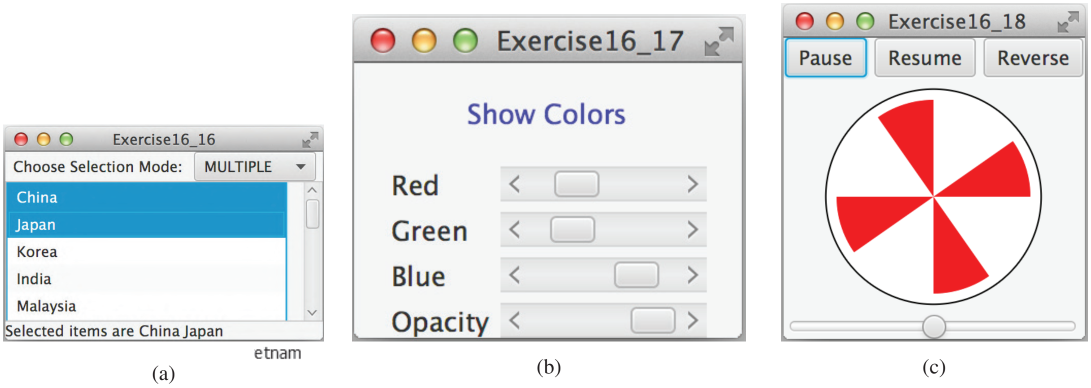
contentDisplay: LEFT graphicTextGap: 20

(a) (b)

FIGURE 16.42 You can dynamically set the font for the message. (b) You can set the alignment and text-position properties of a label dynamically. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****16.15** (*Demonstrate* **`Label`** *properties*) Write a program to let the user dynamically set the properties **`contentDisplay`** and **`graphicTextGap`**, as shown in Figure 16.42b.

***16.16** (*Use* **`ComboBox`** *and* **`ListView`**) Write a program that demonstrates selecting items in a list. The program uses a combo box to specify a selection mode, as shown in Figure 16.43a. When you select items, they are displayed in a label below the list.


Exercise16_16
China
Japan
Korea
India
Malaysia
etnam
Exercise16_17
Show Colors
Red
Green
Blue
Opacity
Exercise16_18
Pause
Resume
Reverse

Choose Selection Mode: MULTIPLE

Selected items are China Japan

(a) (b) (c)

FIGURE 16.43 (a) You can choose single or multiple selection modes in a list. (b) The color changes in the text as you adjust the scroll bars. (c) The program simulates a running fan. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Sections 16.6–16.8

****16.17** (*Use* **`ScrollBar`** *and* **`Slider`**) Write a program that uses scroll bars or sliders to select the color for a text, as shown in Figure 16.43b. Four horizontal scroll bars are used for selecting the colors: red, green, blue, and opacity percentages.

****16.18** (*Simulation: a running fan*) Rewrite Programming Exercise 15.28 to add a slider to control the speed of the fan, as shown in Figure 16.43c.

****16.19** (*Control a group of fans*) Write a program that displays three fans in a group, with control buttons to start and stop all of them, as shown in Figure 16.44.

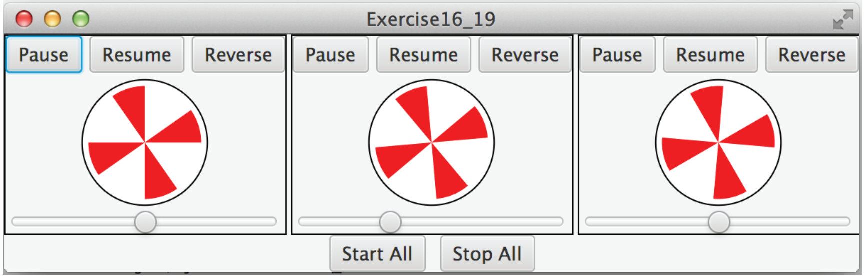
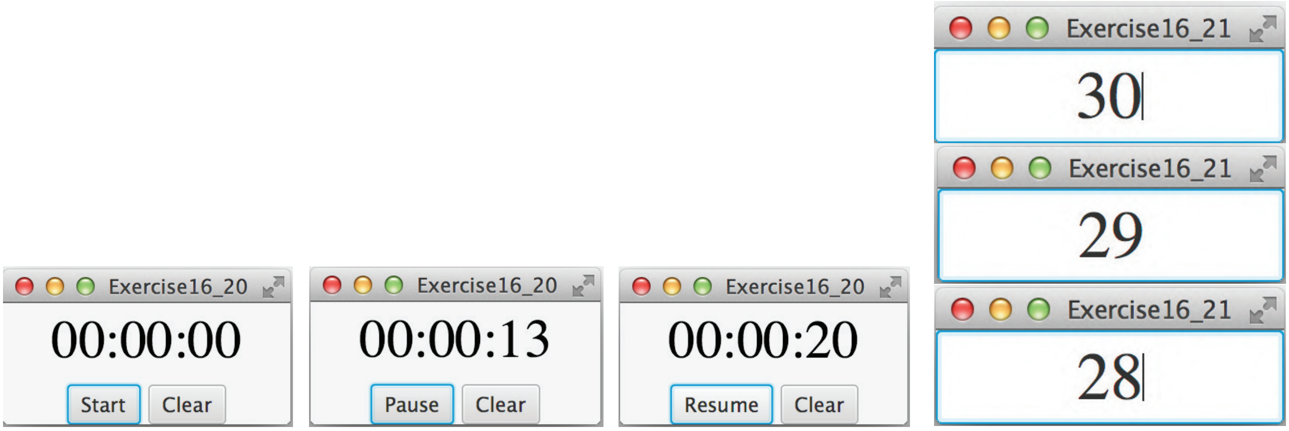

Exercise16_19
Pause
Resume
Reverse
Pause
Resume
Reverse
Pause
Resume
Reverse
Start All
Stop All

FIGURE 16.44 The program runs and controls a group of fans. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***16.20** (*Count-up stopwatch*) Write a program that simulates a stopwatch, as shown in Figure 16.45a. When the user clicks the *Start* button, the button's label is changed to *Pause*, as shown in Figure 16.45b. When the user clicks the *Pause* button, the button's label is changed to *Resume*, as shown in Figure 16.45c. The *Clear* button resets the count to 0 and resets the button's label to *Start*.


Exercise16_20
00:00:00
Start
Clear
Exercise16_20
00:00:13
Pause
Clear
Exercise16_20
00:00:20
Resume
Clear
Exercise16_21
30
Exercise16_21
29
Exercise16_21
28

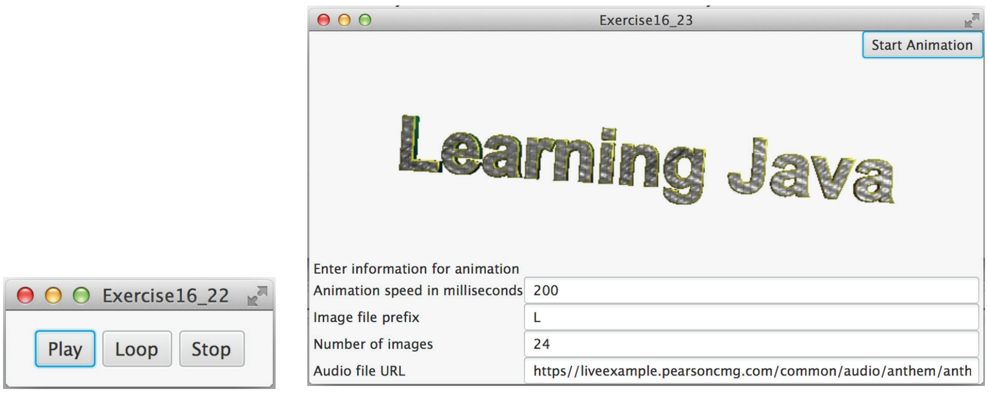
(a) (b) (c) (d)

FIGURE 16.45 (a–c) The program counts up the time. (d) The program counts down the time. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***16.21** *(Count-down stopwatch)* Write a program that allows the user to enter time in seconds in the text field and press the *Enter* key to count down the seconds, as shown in Figure 16.45d. The remaining seconds are redisplayed every second. When the seconds are expired, the program starts to play music continuously.

16.22 *(Play, loop, and stop a sound clip)* Write a program that meets the following requirements:

- Get an audio file from the class directory using `AudioClip`.
- Place three buttons labeled *Play*, *Loop*, and *Stop*, as shown in Figure 16.46a.
- If you click the *Play* button, the audio file is played once. If you click the *Loop* button, the audio file keeps playing repeatedly. If you click the *Stop* button, the playing stops.


Exercise16_22
Play
Loop
Stop
Exercise16_23
Start Animation
Learning Java

Enter information for animation
Animation speed in milliseconds 200
Image file prefix L
Number of images 24
Audio file URL https//liveexample.pearsoncmg.com/common/audio/anthem/anth

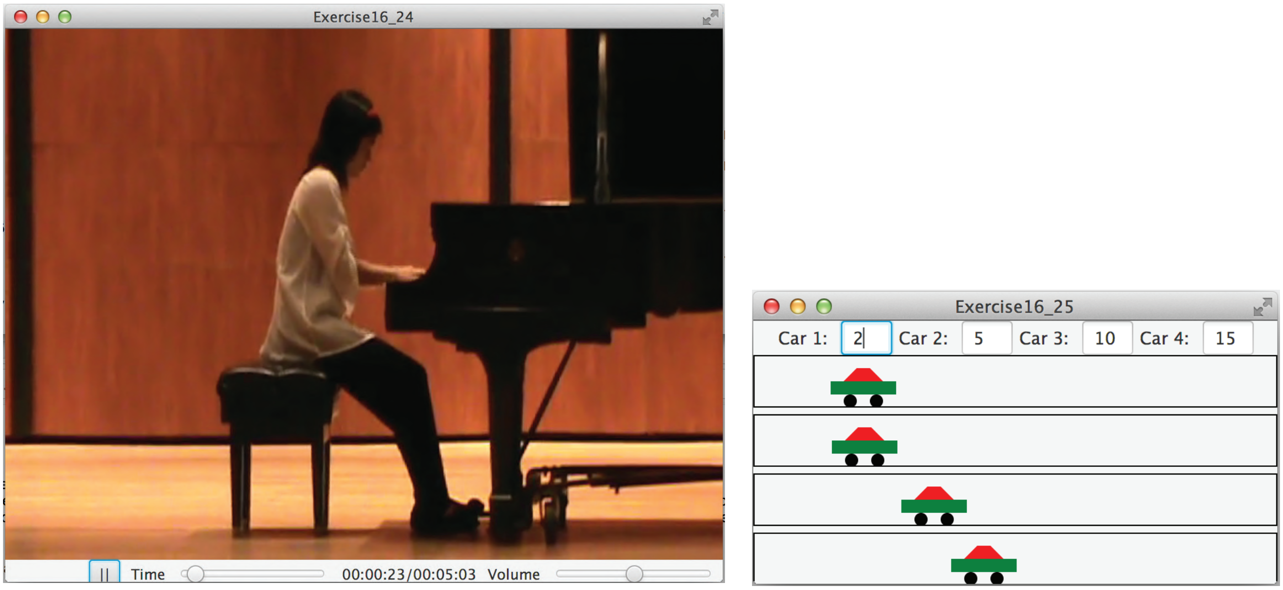
(a) (b)

FIGURE 16.46 (a) Click *Play* to play an audio clip once, click *Loop* to play an audio repeatedly, and click *Stop* to terminate playing. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) The program lets the user specify image files, an audio file, and the animation speed.

****16.23** *(Create an image animator with audio)* Create animation in Figure 16.46b to meet the following requirements:

- Allow the user to specify the animation speed in a text field.
- Get the number of images and image's file-name prefix from the user. For example, if the user enters **n** for the number of images and **L** for the image prefix, then the files are **L1.gif**, **L2.gif**, and so on, to **L*n*.gif**. Assume the images are stored in the **image** directory, a subdirectory of the program's class directory. The animation displays the images one after the other.
- Allow the user to specify an audio file URL. The audio is played while the animation runs.

****16.24** *(Revise Listing 16.14 MediaDemo.java)* Add a slider to enable the user to set the current time for the video and a label to display the current time and the total time for the video. As shown in Figure 16.47a, the total time is 5 minutes and 3 seconds and the current time is 3 minutes and 58 seconds. As the video plays, the slider value and current time are continuously updated.



Exercise16_24

Time 00:00:23/00:00:05:03 Volume

Exercise16_25

Car 1: 2 Car 2: 5 Car 3: 10 Car 4: 15

(a) (b)

Figure 16.47 (a) A slider for current video time and a label to show the current time and total time are added. (b) You can set the speed for each car. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****16.25** *(Racing cars)* Write a program that simulates four cars racing, as shown in Figure 16.47b. You can set the speed for each car, with a maximum of 100.

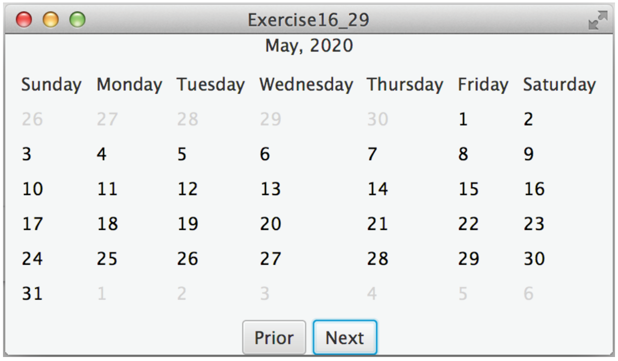
****16.26** *(Simulation: raise flag and play anthem)* Write a program that displays a flag rising up, as shown in Figure 15.15. As the national flag rises, play the national anthem. (You may use a flag image and anthem audio file from Listing 16.15.)

Comprehensive

****16.27** *(Display country flag and flag description)* Listing 16.8, ComboBoxDemo.java, gives a program that lets the user view a country's flag image and description by selecting the country from a combo box. The description is a string coded in the program. Rewrite the program to read the text description from a file. Suppose the descriptions are stored in the files **description0.txt**, . . . , and **description8.txt** under the **text** directory for the nine countries Canada, China, Denmark, France, Germany, India, Norway, the United Kingdom, and the United States, in this order.

****16.28** *(Slide show)* Programming Exercise 15.30 developed a slide show using images. Rewrite that program to develop a slide show using text files. Suppose that 10 text files named **slide0.txt**, **slide1.txt**, . . . , **slide9.txt** are stored in the **text** directory. Each slide displays the text from one file. Each slide is shown for one second, and the slides are displayed in order. When the last slide finishes, the first slide is redisplayed, and so on. Use a text area to display the slide.

*****16.29** *(Display a calendar)* Write a program that displays the calendar for the current month. You can use the *Prior* and *Next* buttons to show the calendar of the previous or next month. Display the dates in the current month in black and display the dates in the previous month and next month in gray, as shown in Figure 16.48.

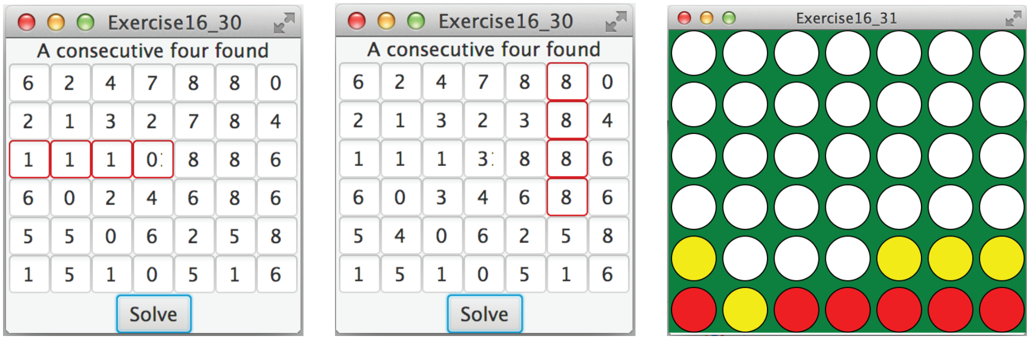


May, 2020

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
26	27	28	29	30	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

FIGURE 16.48 The program displays the calendar for the current month. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****16.30** *(Pattern recognition: consecutive four equal numbers)* Write a GUI program for Programming Exercise 8.19, as shown in Figures 16.49a–b. Let the user enter the numbers in the text fields in a grid of 6 rows and 7 columns. The user can click the *Solve* button to highlight a sequence of four equal numbers, if it exists. Initially, the values in the text fields are randomly filled with numbers from 0 to 9.



A consecutive four found

6	2	4	7	8	8	0
2	1	3	2	7	8	4
1	1	1	0	8	8	6
6	0	2	4	6	8	6
5	5	0	6	2	5	8
1	5	1	0	5	1	6

A consecutive four found

6	2	4	7	8	8	0
2	1	3	2	3	8	4
1	1	1	3	8	8	6
6	0	3	4	6	8	6
5	4	0	6	2	5	8
1	5	1	0	5	1	6

(a) (b) (c)

FIGURE 16.49 (a and b) Clicking the *Solve* button highlights the four consecutive numbers in a row, a column, or a diagonal. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (c) The program enables two players to play the connect-four game.

*****16.31** *(Game: connect four)* Programming Exercise 8.20 enables two players to play the connect-four game on the console. Rewrite a GUI version for the program, as shown in Figure 16.49c. The program enables two players to place red and yellow discs in turn. To place a disk, the player needs to click an available cell. An *available cell* is unoccupied and its downward neighbor is occupied. The program flashes the four winning cells if a player wins, and reports no winners if all cells are occupied with no winners.

CHAPTER 17

BINARY I/O

Objectives

- To discover how I/O is processed in Java (§17.2).
- To distinguish between text I/O and binary I/O (§17.3).
- To read and write bytes using **`FileInputStream`** and **`FileOutputStream`** (§17.4.1).
- To filter data using the base classes **`FilterInputStream`** and **`FilterOutputStream`** (§17.4.2).
- To read and write primitive values and strings using **`DataInputStream`** and **`DataOutputStream`** (§17.4.3).
- To improve I/O performance by using **`BufferedInputStream`** and **`BufferedOutputStream`** (§17.4.4).
- To write a program that copies a file (§17.5).
- To store and restore objects using **`ObjectOutputStream`** and **`ObjectInputStream`** (§17.6).
- To implement the **`Serializable`** interface to make objects serializable (§17.6.1).
- To serialize arrays (§17.6.2).
- To read and write files using the **`RandomAccessFile`** class (§17.7).

17.1 Introduction

Java provides many classes for performing text I/O and binary I/O.

Key Point

text file

binary file

Files can be classified as either text or binary. A file that can be processed (read, created, or modified) using a text editor such as Notepad on Windows or vi on UNIX is called a *text file*. All other files are called *binary files*. You cannot read binary files using a text editor—they are designed to be read by programs. For example, Java source programs are text files and can be read by a text editor, but Java class files are binary files and are read by the JVM.

why binary I/O?

Although it is not technically precise and correct, you can envision a text file as consisting of a sequence of characters, and a binary file as consisting of a sequence of bits. Characters in a text file are encoded using a character-encoding scheme such as ASCII or Unicode. For example, the decimal integer `199` is stored as a sequence of three characters `199` in a text file, and the same integer is stored as a byte-type value `C7` in a binary file, because decimal `199` equals hex `C7` ($199 = 12 \times 16^1 + 7$). The advantage of binary files is that they are more efficient to process than text files.

text I/O

binary I/O

Java offers many classes for performing file input and output. These can be categorized as *text I/O classes* and *binary I/O classes*. In Section 12.11, File Input and Output, you learned how to read and write strings and numeric values from/to a text file using `Scanner` and `PrintWriter`. This chapter introduces the classes for performing binary I/O.

17.2 How Is Text I/O Handled in Java?

Text data are read using the `Scanner` *class and written using the* `PrintWriter` *class.*

Key Point

Recall that a `File` object encapsulates the properties of a file or a path but does not contain the methods for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. For example, to write text to a file named **temp.txt**, you can create an object using the `PrintWriter` class as follows:

```
PrintWriter output = new PrintWriter("temp.txt");
```

You can now invoke the `print` method on the object to write a string to the file. For example, the following statement writes `Java 101` to the file:

```
output.print("Java 101");
```

The following statement closes the file:

```
output.close();
```

There are many I/O classes for various purposes. In general, these can be classified as input classes and output classes. An *input class* contains the methods to read data, and an *output class* contains the methods to write data. `PrintWriter` is an example of an output class, and `Scanner` is an example of an input class. The following code creates an input object for the file **temp.txt** and reads data from the file:

```
Scanner input = new Scanner(new File("temp.txt"));
System.out.println(input.nextLine());
```

If **temp.txt** contains the text `Java 101`, `input.nextLine()` returns the string `"Java 101"`.

stream

input stream

output stream

Figure 17.1 illustrates Java I/O programming. An input object reads a *stream* of data from a file, and an output object writes a stream of data to a file. An input object is also called an *input stream* and an output object an *output stream*.

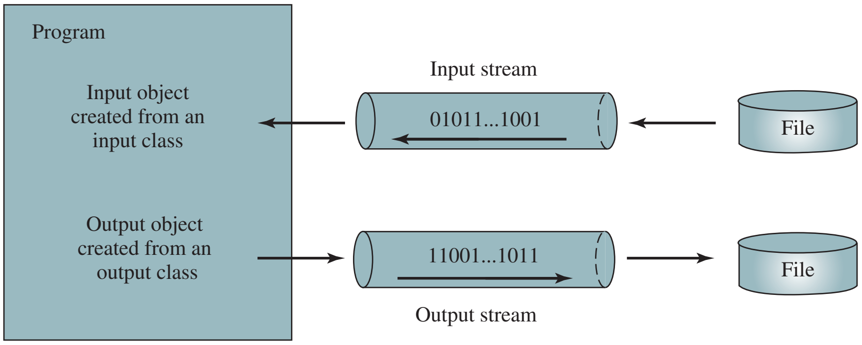

Program
Input object created from an input class
Output object created from an output class
Input stream
01011...1001
File
11001...1011
Output stream
File

FIGURE 17.1 The program receives data through an input object and sends data through an output object.

Check Point

17.2.1 What is a text file and what is a binary file? Can you view a text file or a binary file using a text editor?

17.2.2 How do you read or write text data in Java? What is a stream?

17.3 Text I/O vs. Binary I/O

Binary I/O does not involve encoding or decoding and thus is more efficient than text I/O.

Key Point

Computers do not differentiate between binary files and text files. All files are stored in binary format, and thus all files are essentially binary files. Text I/O is built upon binary I/O to provide a level of abstraction for character encoding and decoding, as shown in Figure 17.2a. Encoding and decoding are automatically performed for text I/O. The JVM converts Unicode to a file-specific encoding when writing a character, and converts a file-specific encoding to Unicode when reading a character. For example, suppose you write the string `"199"` using text I/O to a file, each character is written to the file. Since the Unicode for character `1` is `0x0031`, the Unicode `0x0031` is converted to a code that depends on the encoding scheme for the file. (Note the prefix `0x` denotes a hex number.) In the United States, the default encoding for text files on Windows is ASCII. The ASCII code for character `1` is `49` (`0x31` in hex) and for

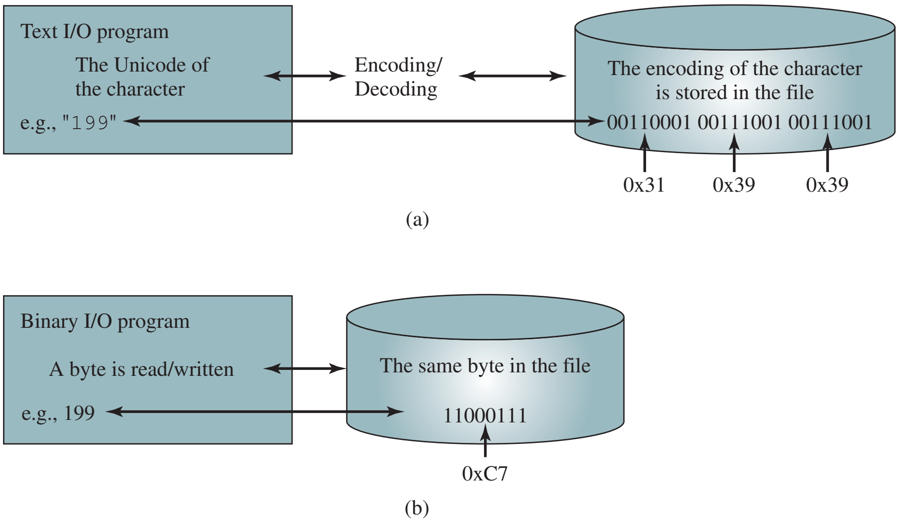

Text I/O program
The Unicode of the character
e.g., "199"
Encoding/ Decoding
The encoding of the character is stored in the file
00110001 00111001 00111001
0x31
0x39
0x39
(a)
Binary I/O program
A byte is read/written
e.g., 199
The same byte in the file
11000111
0xC7
(b)

FIGURE 17.2 Text I/O requires encoding and decoding, whereas binary I/O does not.

character `9` is `57` (`0x39` in hex). Thus, to write the characters `199`, three bytes—`0x31`, `0x39`, and `0x39`—are sent to the output, as shown in Figure 17.2a.

Binary I/O does not require conversions. If you write a numeric value to a file using binary I/O, the exact value in the memory is copied into the file. For example, a byte-type value `199` is represented as `0xC7` ($199 = 12 \times 16^1 + 7$) in the memory and appears exactly as `0xC7` in the file, as shown in Figure 17.2b. When you read a byte using binary I/O, one byte value is read from the input.

In general, you should use text input to read a file created by a text editor or a text output program, and use binary input to read a file created by a Java binary output program.

Binary I/O is more efficient than text I/O because binary I/O does not require encoding and decoding. Binary files are independent of the encoding scheme on the host machine and thus are portable. Java programs on any machine can read a binary file created by a Java program. This is why Java class files are binary files. Java class files can run on a JVM on any machine.

.txt and .dat

Note

For consistency, this book uses the extension **.txt** to name text files and **.dat** to name binary files.

Check Point

17.3.1 What are the differences between text I/O and binary I/O?

17.3.2 How is a Java character represented in the memory, and how is a character represented in a text file?

17.3.3 If you write the string `"ABC"` to an ASCII text file, what values are stored in the file?

17.3.4 If you write the string `"100"` to an ASCII text file, what values are stored in the file? If you write a numeric byte-type value `100` using binary I/O, what values are stored in the file?

17.3.5 What is the encoding scheme for representing a character in a Java program? By default, what is the encoding scheme for a text file on Windows?

17.4 Binary I/O Classes

Key Point

The abstract `InputStream` *is the root class for reading binary data, and the abstract* `OutputStream` *is the root class for writing binary data.*

The design of the Java I/O classes is a good example of applying inheritance, where common operations are generalized in superclasses, and subclasses provide specialized operations. Figure 17.3 lists some of the classes for performing binary I/O. `InputStream` is the root for

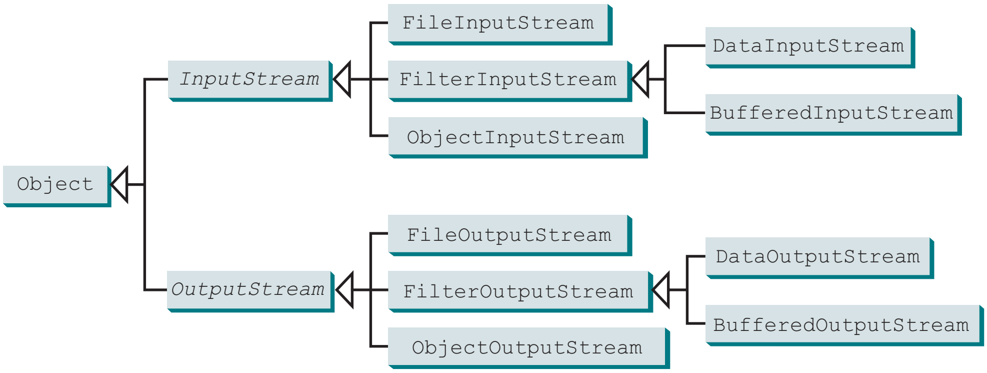


FIGURE 17.3 `InputStream`, `OutputStream`, and their subclasses are for performing binary I/O.

`java.io.InputStream`	
`+read(): int`	Reads the next byte of data from the input stream. The value byte is returned as an `int` value in the range `0–255`. If no byte is available because the end of the stream has been reached, the value –1 is returned.
`+read(b: byte[]): int`	Reads up to `b.length` bytes into array `b` from the input stream and returns the actual number of bytes read. Returns –1 at the end of the stream.
`+read(b: byte[], off: int, len: int): int`	Reads bytes from the input stream and stores them in `b[off]`, `b[off+1]`, `...`, `b[off+len-1]`. The actual number of bytes read is returned. Returns `-1` at the end of the stream.
`+close(): void`	Closes this input stream and releases any system resources occupied by it.
`+skip(n: long): long`	Skips over and discards `n` bytes of data from this input stream. The actual number of bytes skipped is returned.

FIGURE 17.4 The abstract **`InputStream`** class defines the methods for the input stream of bytes.

binary input classes, and **`OutputStream`** is the root for binary output classes. Figures 17.4 and 17.5 list all the methods in the classes **`InputStream`** and **`OutputStream`**.

Note

All the methods in the binary I/O classes are declared to throw **`java.io.IOException`** or a subclass of **`java.io.IOException`**.

`throws IOException`

`java.io.OutputStream`	
`+write(int b): void`	Writes the specified byte to this output stream. The parameter `b` is an `int` value. `(byte)b` is written to the output stream.
`+write(b: byte[]): void`	Writes all the bytes in array `b` to the output stream.
`+write(b: byte[], off: int, len: int): void`	Writes `b[off]`, `b[off+1]`, `...`, `b[off+len-1]` into the output stream.
`+close(): void`	Closes this output stream and releases any system resources occupied by it.
`+flush(): void`	Flushes this output stream and forces any buffered output bytes to be written out.

FIGURE 17.5 The abstract **`OutputStream`** class defines the methods for the output stream of bytes.

17.4.1 `FileInputStream/FileOutputStream`

`FileInputStream`/**`FileOutputStream`** are for reading/writing bytes from/to files. All the methods in these classes are inherited from **`InputStream`** and **`OutputStream`**. **`FileInputStream`**/**`FileOutputStream`** do not introduce new methods. To construct a **`FileInputStream`**, use the constructors shown in Figure 17.6.

A **`java.io.FileNotFoundException`** will occur if you attempt to create a **`FileInputStream`** with a nonexistent file.

`FileNotFoundException`

To construct a **`FileOutputStream`**, use the constructors shown in Figure 17.7.

If the file does not exist, a new file will be created. If the file already exists, the first two constructors will delete the current content of the file. To retain the current content and append new data into the file, use the last two constructors and pass **`true`** to the **`append`** parameter.

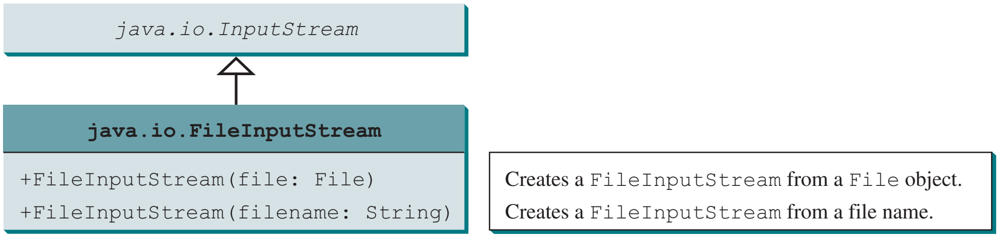

java.io.InputStream
java.io.FileInputStream
+FileInputStream(file: File)
+FileInputStream(filename: String)
Creates a FileInputStream from a File object.
Creates a FileInputStream from a file name.

Figure 17.6 `FileInputStream` inputs a stream of bytes from a file.

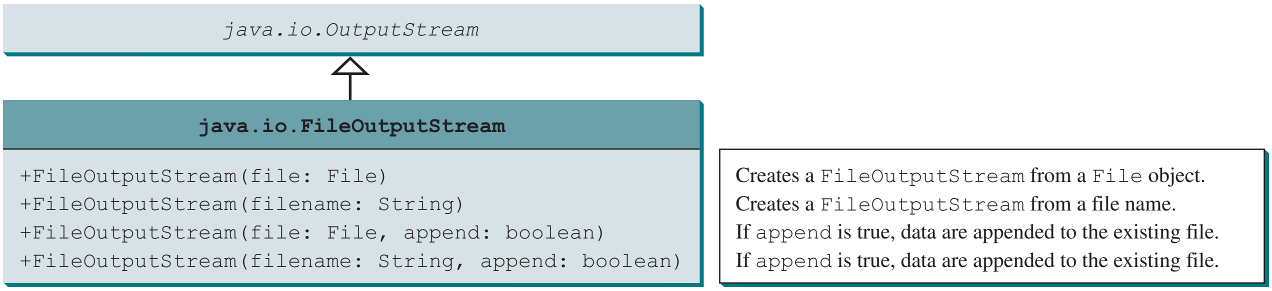

java.io.OutputStream
java.io.FileOutputStream
+FileOutputStream(file: File)
+FileOutputStream(filename: String)
+FileOutputStream(file: File, append: boolean)
+FileOutputStream(filename: String, append: boolean)
Creates a FileOutputStream from a File object.
Creates a FileOutputStream from a file name.
If append is true, data are appended to the existing file.
If append is true, data are appended to the existing file.

Figure 17.7 `FileOutputStream` outputs a stream of bytes to a file.

`IOException`

Almost all the methods in the I/O classes throw `java.io.IOException`. Therefore, you have to declare to throw `java.io.IOException` in the method in (a) or place the code in a try-catch block in (b), as shown below:

(a)
Declaring exception in the method

```
public static void main(String[] args)
    throws IOException {
  // Perform I/O operations
}
```

(b)
Using try-catch block

```
public static void main(String[] args) {
  try {
    // Perform I/O operations
  }
  catch (IOException ex) {
    ex.printStackTrace();
  }
}
```

Listing 17.1 uses binary I/O to write 10 byte values from `1` to `10` to a file named **temp.dat** and reads them back from the file.

Listing 17.1 TestFileStream.java

import

output stream

output

```
import java.io.*;

public class TestFileStream {
  public static void main(String[] args) throws IOException {
    try (
      // Create an output stream to the file
      FileOutputStream output = new FileOutputStream("temp.dat");
    ) {
      // Output values to the file
      for (int i = 1; i <= 10; i++)
        output.write(i);
    }

    try (
15        // Create an input stream for the file
16        FileInputStream input = new FileInputStream("temp.dat");
17      ) {
18        // Read values from the file
19        int value;
20        while ((value = input.read()) != -1)
21          System.out.print(value + " ");
22      }
23    }
24  }
```

input stream

input

```
1 2 3 4 5 6 7 8 9 10
```

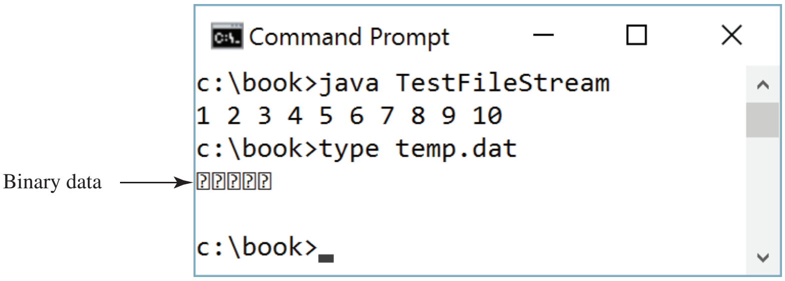
AutoClosable

The program uses the try-with-resources to declare and create input and output streams so they will be automatically closed after they are used. The `java.io.InputStream` and `java.io.OutputStream` classes implement the `AutoClosable` interface. The `AutoClosable` interface defines the `close()` method that closes a resource. Any object of the `AutoClosable` type can be used with the try-with-resources syntax for automatic closing.

A `FileOutputStream` is created for the file **temp.dat** in line 7. The `for` loop writes 10 byte values into the file (lines 10 and 11). Invoking `write(i)` is the same as invoking `write((byte)i)`. Line 16 creates a `FileInputStream` for the file **temp.dat**. Values are read from the file and displayed on the console in lines 19–21. The expression `((value = input.read()) != -1)` (line 20) reads a byte from `input.read()`, assigns it to `value`, and checks whether it is `-1`. The input value of `-1` signifies the end of a file.

end of a file

The file **temp.dat** created in this example is a binary file. It can be read from a Java program but not from a text editor, as shown in Figure 17.8.



```
c:\book>java TestFileStream
1 2 3 4 5 6 7 8 9 10
c:\book>type temp.dat
□□□□□

c:\book>_
```

Figure 17.8 A binary file cannot be displayed in text mode. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Tip

When a stream is no longer needed, always close it using the `close()` method or automatically close it using a try-with-resource statement. Not closing streams may cause data corruption in the output file or other programming errors.

close stream

Note

The root directory for the file is the classpath directory. For the example in this book, the root directory is **c:\book**, so the file **temp.dat** is located at **c:\book**. If you wish to place **temp.dat** in a specific directory, replace line 6 with

where is the file?

```
FileOutputStream output =
  new FileOutputStream ("directory/temp.dat");
```

Note

An instance of `FileInputStream` can be used as an argument to construct a `Scanner`, and an instance of `FileOutputStream` can be used as an argument to construct a `PrintWriter`. You can create a `PrintWriter` to append text into a file using

appending to text file

```
new PrintWriter(new FileOutputStream("temp.txt", true));
```

If **temp.txt** does not exist, it is created. If **temp.txt** already exists, new data are appended to the file. See Programming Exercise 17.1.

17.4.2 `FilterInputStream`/`FilterOutputStream`

Filter streams are streams that filter bytes for some purpose. The basic byte input stream provides a `read` method that can be used only for reading bytes. If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream. Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters. `FilterInputStream` and `FilterOutputStream` are the base classes for filtering data. When you need to process primitive numeric types, use `DataInputStream` and `DataOutputStream` to filter bytes.

17.4.3 `DataInputStream`/`DataOutputStream`

`DataInputStream` reads bytes from the stream and converts them into appropriate primitive-type values or strings. `DataOutputStream` converts primitive-type values or strings into bytes and outputs the bytes to the stream.

`DataInputStream` extends `FilterInputStream` and implements the `DataInput` interface, as shown in Figure 17.9. `DataOutputStream` extends `FilterOutputStream` and implements the `DataOutput` interface, as shown in Figure 17.10.

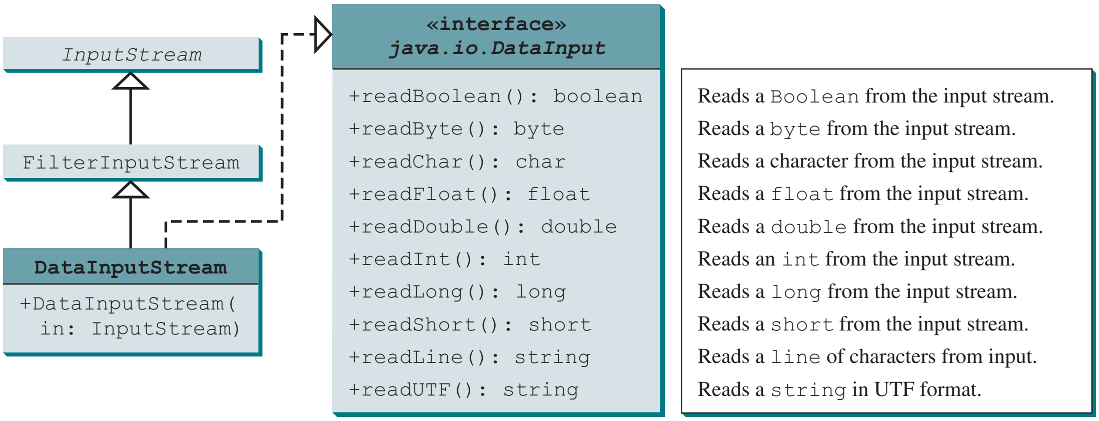

InputStream
FilterInputStream
DataInputStream
+DataInputStream(
in: InputStream)
«interface»
java.io.DataInput
+readBoolean(): boolean
+readByte(): byte
+readChar(): char
+readFloat(): float
+readDouble(): double
+readInt(): int
+readLong(): long
+readShort(): short
+readLine(): string
+readUTF(): string
Reads a Boolean from the input stream.
Reads a byte from the input stream.
Reads a character from the input stream.
Reads a float from the input stream.
Reads a double from the input stream.
Reads an int from the input stream.
Reads a long from the input stream.
Reads a short from the input stream.
Reads a line of characters from input.
Reads a string in UTF format.

FIGURE 17.9 `DataInputStream` filters an input stream of bytes into primitive data-type values and strings.

`DataInputStream` implements the methods defined in the `DataInput` interface to read primitive data-type values and strings. `DataOutputStream` implements the methods defined in the `DataOutput` interface to write primitive data-type values and strings. Primitive values are copied from memory to the output without any conversions. Characters in a string may be written in several ways, as discussed in the next section.

Characters and Strings in Binary I/O

A Unicode character consists of two bytes. The `writeChar(char c)` method writes the Unicode of character `c` to the output. The `writeChars(String s)` method writes the Unicode for each character in the string `s` to the output. The `writeBytes(String s)` method writes the lower byte of the Unicode for each character in the string `s` to the output. The high byte of the Unicode is discarded. The `writeBytes` method is suitable for strings that consist of

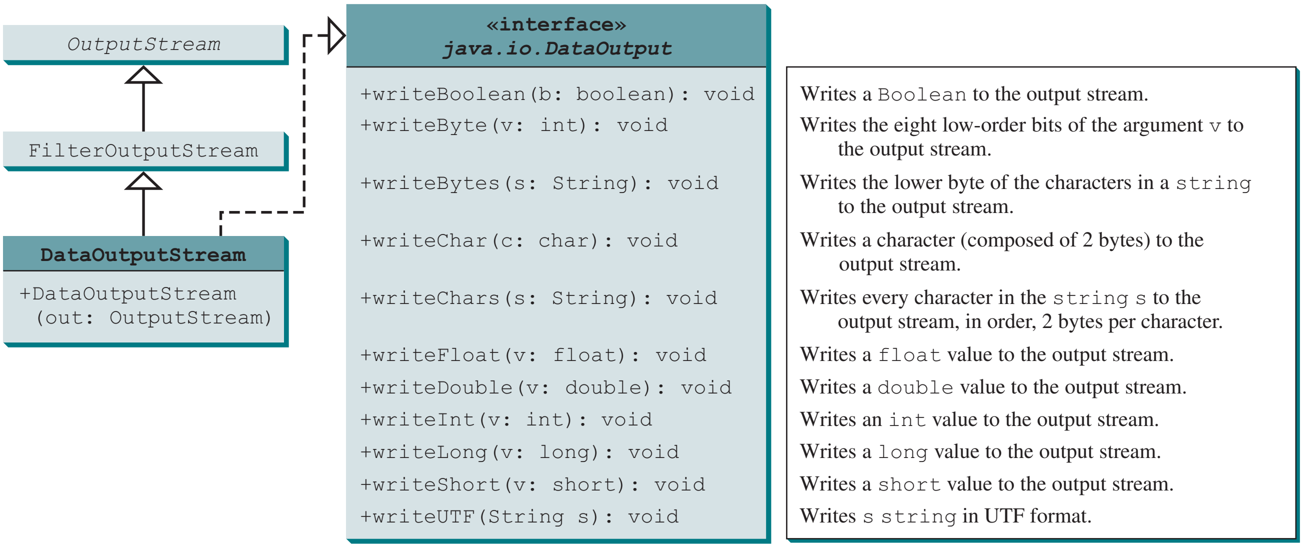

OutputStream
FilterOutputStream
DataOutputStream
+DataOutputStream (out: OutputStream)
«interface» java.io.DataOutput
+writeBoolean(b: boolean): void
+writeByte(v: int): void
+writeBytes(s: String): void
+writeChar(c: char): void
+writeChars(s: String): void
+writeFloat(v: float): void
+writeDouble(v: double): void
+writeInt(v: int): void
+writeLong(v: long): void
+writeShort(v: short): void
+writeUTF(String s): void
Writes a Boolean to the output stream.
Writes the eight low-order bits of the argument v to the output stream.
Writes the lower byte of the characters in a string to the output stream.
Writes a character (composed of 2 bytes) to the output stream.
Writes every character in the string s to the output stream, in order, 2 bytes per character.
Writes a float value to the output stream.
Writes a double value to the output stream.
Writes an int value to the output stream.
Writes a long value to the output stream.
Writes a short value to the output stream.
Writes s string in UTF format.

FIGURE 17.10 `DataOutputStream` enables you to write primitive data-type values and strings into an output stream.

ASCII characters, since an ASCII code is stored only in the lower byte of a Unicode. If a string consists of non-ASCII characters, you have to use the `writeChars` method to write the string.

The `writeUTF(String s)` method writes a string using the UTF coding scheme. UTF is efficient for compressing a string with Unicode characters. For more information on UTF, see Supplement III.Z, UTF in Java. The `readUTF()` method reads a string that has been written using the `writeUTF` method.

Creating DataInputStream/DataOutputStream

`DataInputStream`/`DataOutputStream` are created using the following constructors (see Figures 17.9 and 17.10):

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream outstream)
```

The following statements create data streams. The first statement creates an input stream for the file **in.dat**; the second statement creates an output stream for the file **out.dat**.

```
DataInputStream input =
  new DataInputStream(new FileInputStream("in.dat"));
DataOutputStream output =
  new DataOutputStream(new FileOutputStream("out.dat"));
```

Listing 17.2 writes student names and scores to a file named **temp.dat** and reads the data back from the file.

LISTING 17.2 TestDataStream.java

```
import java.io.*;

public class TestDataStream {
  public static void main(String[] args) throws IOException {
    try ( // Create an output stream for file temp.dat
      DataOutputStream output =
        new DataOutputStream(new FileOutputStream("temp.dat"));
    ) {
```

output stream

output

input stream

input

```
 9        // Write student test scores to the file
10        output.writeUTF("John");
11        output.writeDouble(85.5);
12        output.writeUTF("Susan");
13        output.writeDouble(185.5);
14        output.writeUTF("Kim");
15        output.writeDouble(105.25);
16      }
17
18      try ( // Create an input stream for file temp.dat
19        DataInputStream input =
20          new DataInputStream(new FileInputStream("temp.dat"));
21      ) {
22        // Read student test scores from the file
23        System.out.println(input.readUTF() + " " + input.readDouble());
24        System.out.println(input.readUTF() + " " + input.readDouble());
25        System.out.println(input.readUTF() + " " + input.readDouble());
26      }
27    }
28  }
```

```
John 85.5
Susan 185.5
Kim 105.25
```

A `DataOutputStream` is created for file **temp.dat** in lines 6 and 7. Student names and scores are written to the file in lines 10–15. A `DataInputStream` is created for the same file in lines 19 and 20. Student names and scores are read back from the file and displayed on the console in lines 23–25.

`DataInputStream` and `DataOutputStream` read and write Java primitive-type values and strings in a machine-independent fashion, thereby enabling you to write a data file on one machine and read it on another machine that has a different operating system or file structure. An application uses a data output stream to write data that can later be read by a program using a data input stream.

`DataInputStream` filters data from an input stream into appropriate primitive-type values or strings. `DataOutputStream` converts primitive-type values or strings into bytes and outputs the bytes to an output stream. You can view `DataInputStream`/`FileInputStream` and `DataOutputStream`/`FileOutputStream` working in a pipe line as shown in Figure 17.11.

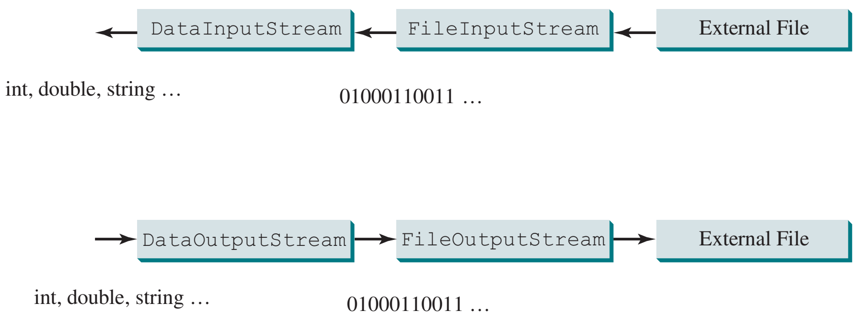


FIGURE 17.11 `DataInputStream` filters an input stream of byte to data and `DataOutputStream` converts data into a stream of bytes.

Caution

You have to read data in the same order and format in which they are stored. For example, since names are written in UTF using `writeUTF`, you must read names using `readUTF`.

Detecting the End of a File

If you keep reading data at the end of an `InputStream`, an `EOFException` will occur. This exception can be used to detect the end of a file, as shown in Listing 17.3.

EOFException

LISTING 17.3 DetectEndOfFile.java

```
 1  import java.io.*;
 2
 3  public class DetectEndOfFile {
 4    public static void main(String[] args) {
 5      try {
 6        try (DataOutputStream output =
 7          new DataOutputStream(new FileOutputStream("test.dat"))) {
 8          output.writeDouble(4.5);
 9          output.writeDouble(43.25);
10          output.writeDouble(3.2);
11        }
12
13        try (DataInputStream input =
14          new DataInputStream(new FileInputStream("test.dat"))) {
15          while (true)
16            System.out.println(input.readDouble());
17        }
18      }
19      catch (EOFException ex) {
20        System.out.println("All data were read");
21      }
22      catch (IOException ex) {
23        ex.printStackTrace();
24      }
25    }
26  }
```

output stream

output

input stream

input

EOFException

```
4.5
43.25
3.2
All data were read
```

The program writes three double values to the file using `DataOutputStream` (lines 6–11) and reads the data using `DataInputStream` (lines 13–17). When reading past the end of the file, an `EOFException` is thrown. The exception is caught in line 19.

17.4.4 BufferedInputStream/BufferedOutputStream

`BufferedInputStream`/`BufferedOutputStream` can be used to speed up input and output by reducing the number of disk reads and writes. Using `BufferedInputStream`, the whole block of data on the disk is read into the buffer in the memory once. The individual data are then loaded to your program from the buffer, as shown in Figure 17.12a. Using `BufferedOutputStream`, the individual data are first written to the buffer in the memory. When the buffer is full, all data in the buffer are written to the disk once, as shown in Figure 17.12b.

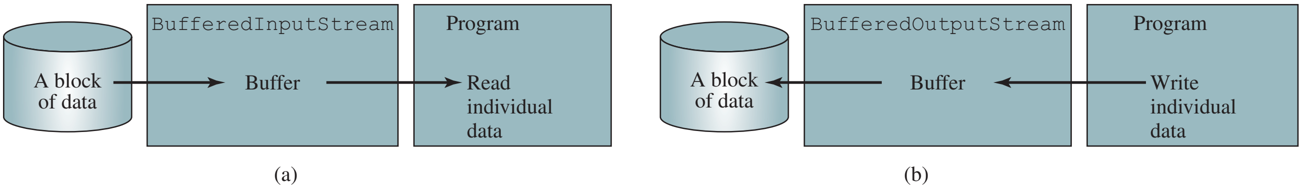
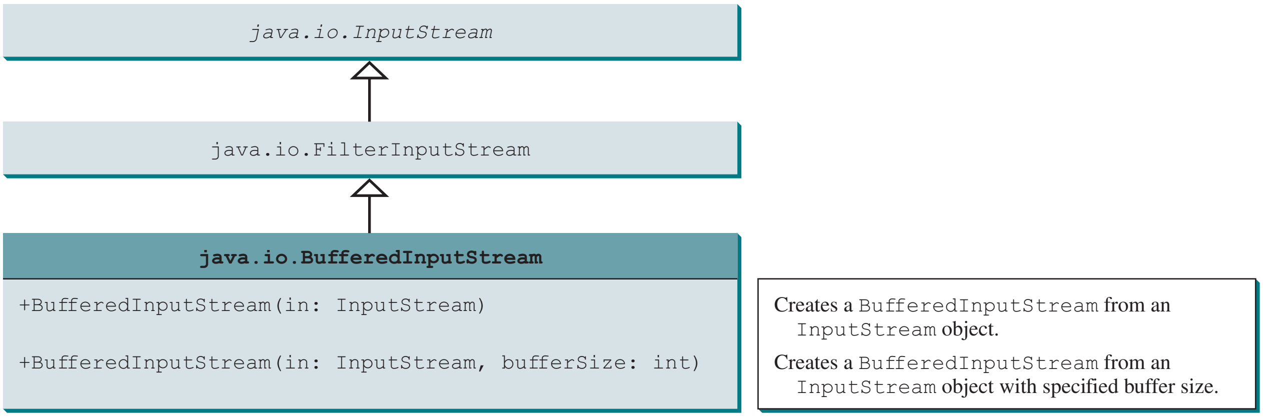

A block of data
BufferedInputStream
Buffer
Program
Read individual data
(a)
A block of data
BufferedOutputStream
Buffer
Program
Write individual data
(b)

FIGURE 17.12 Buffer I/O places data in a buffer for fast processing.

`BufferedInputStream`/**`BufferedOutputStream`** does not contain new methods. All the methods in **`BufferedInputStream`**/**`BufferedOutputStream`** are inherited from the **`InputStream`**/**`OutputStream`** classes. **`BufferedInputStream`**/**`BufferedOutputStream`** manages a buffer behind the scene and automatically reads/writes data from/to disk on demand.

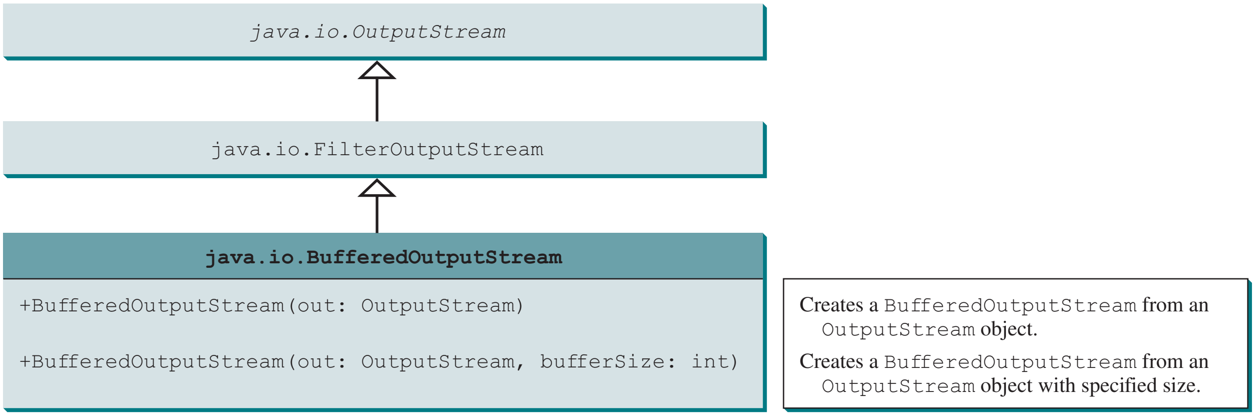
You can wrap a **`BufferedInputStream`**/**`BufferedOutputStream`** on any **`InputStream`**/**`OutputStream`** using the constructors shown in Figures 17.13 and 17.14.


java.io.InputStream
java.io.FilterInputStream
java.io.BufferedInputStream
+BufferedInputStream(in: InputStream)
+BufferedInputStream(in: InputStream, bufferSize: int)

Creates a `BufferedInputStream` from an `InputStream` object.

Creates a `BufferedInputStream` from an `InputStream` object with specified buffer size.

FIGURE 17.13 **`BufferedInputStream`** buffers an input stream.


java.io.OutputStream
java.io.FilterOutputStream
java.io.BufferedOutputStream
+BufferedOutputStream(out: OutputStream)
+BufferedOutputStream(out: OutputStream, bufferSize: int)

Creates a `BufferedOutputStream` from an `OutputStream` object.

Creates a `BufferedOutputStream` from an `OutputStream` object with specified size.

FIGURE 17.14 **`BufferedOutputStream`** buffers an output stream.

If no buffer size is specified, the default size is `512` bytes. You can improve the performance of the `TestDataStream` program in Listing 17.2 by adding buffers in the stream in lines 6–9 and 19–20, as follows:

```
DataOutputStream output = new DataOutputStream(
  new BufferedOutputStream(new FileOutputStream("temp.dat")));

DataInputStream input = new DataInputStream(
  new BufferedInputStream(new FileInputStream("temp.dat")));
```

Tip

You should always use buffered I/O to speed up input and output. For small files, you may not notice performance improvements. However, for large files—over 100 MB—you will see substantial improvements using buffered I/O.

Check Point

17.4.1 The `read()` method in `InputStream` reads a byte. Why does it return an `int` instead of a `byte`? Find the abstract methods in `InputStream` and `OutputStream`.

17.4.2 Why do you have to declare to throw `IOException` in the method or use a try-catch block to handle `IOException` for Java I/O programs?

17.4.3 Why should you always close streams? How do you close streams?

17.4.4 Does `FileInputStream`/`FileOutputStream` introduce any new methods beyond the methods inherited from `InputStream`/`OutputStream`? How do you create a `FileInputStream`/`FileOutputStream`?

17.4.5 What will happen if you attempt to create an input stream on a nonexistent file? What will happen if you attempt to create an output stream on an existing file? Can you append data to an existing file?

17.4.6 How do you append data to an existing text file using `java.io.PrintWriter`?

17.4.7 What is written to a file using `writeByte(91)` on a `FileOutputStream`?

17.4.8 What is wrong in the following code?

```
import java.io.*;

public class Test {
  public static void main(String[] args) {
    try (
      FileInputStream fis = new FileInputStream("test.dat"); ) {
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
    catch (FileNotFoundException ex) {
      ex.printStackTrace();
    }
  }
}
```

17.4.9 Suppose a file contains an unspecified number of `double` values that were written to the file using the `writeDouble` method using a `DataOutputStream`. How do you write a program to read all these values? How do you detect the end of a file?

17.4.10 How do you check the end of a file in an input stream (`FileInputStream`, `DataInputStream`)?

17.4.11 Suppose you run the following program on Windows using the default ASCII encoding after the program is finished. How many bytes are there in the file **t.txt**? Show the contents of each byte.

```
public class Test {
  public static void main(String[] args)
      throws java.io.IOException {
    try (java.io.PrintWriter output =
        new java.io.PrintWriter("t.txt"); ) {
      output.printf("%s", "1234");
      output.printf("%s", "5678");
      output.close();
    }
  }
}
```

17.4.12 After the following program is finished, how many bytes are there in the file **t.dat**? Show the contents of each byte.

```
import java.io.*;

public class Test {
  public static void main(String[] args) throws IOException {
    try (DataOutputStream output = new DataOutputStream(
        new FileOutputStream("t.dat")); ) {
      output.writeInt(1234);
      output.writeInt(5678);
      output.close();
    }
  }
}
```

17.4.13 For each of the following statements on a `DataOutputStream output`, how many bytes are sent to the output?

```
output.writeChar('A');
output.writeChars("BC");
output.writeUTF("DEF");
```

17.4.14 What are the advantages of using buffered streams? Are the following statements correct?

```
BufferedInputStream input1 =
  new BufferedInputStream(new FileInputStream("t.dat"));

DataInputStream input2 = new DataInputStream(
  new BufferedInputStream(new FileInputStream("t.dat")));

DataOutputStream output = new DataOutputStream(
  new BufferedOutputStream(new FileOutnputStream("t.dat")));
```

17.5 Case Study: Copying Files

Key Point

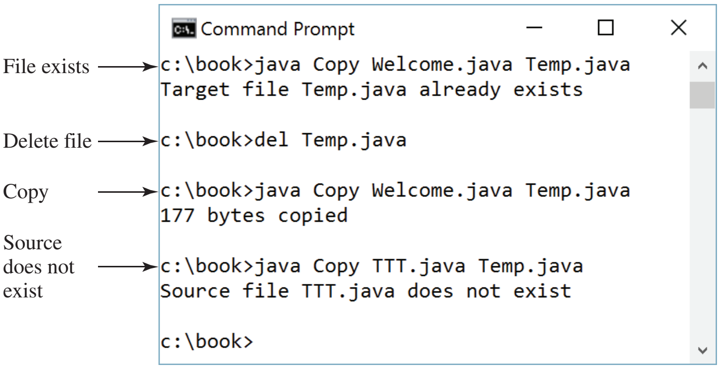
This section develops a useful utility for copying files.

VideoNote
Copy file

In this section, you will learn how to write a program that lets users copy files. The user needs to provide a source file and a target file as command-line arguments using the command

```
java Copy source target
```

The program copies the source file to the target file and displays the number of bytes in the file. The program should alert the user if the source file does not exist or if the target file already exists. A sample run of the program is shown in Figure 17.15.



```
c:\book>java Copy Welcome.java Temp.java
Target file Temp.java already exists

c:\book>del Temp.java

c:\book>java Copy Welcome.java Temp.java
177 bytes copied

c:\book>java Copy TTT.java Temp.java
Source file TTT.java does not exist

c:\book>
```

FIGURE 17.15 The program copies a file. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

To copy the contents from a source file to a target file, it is appropriate to use an input stream to read bytes from the source file, and an output stream to send bytes to the target file, regardless of the file's contents. The source file and the target file are specified from the command line. Create an `InputFileStream` for the source file, and an `OutputFileStream` for the target file. Use the `read()` method to read a byte from the input stream and then use the `write(b)` method to write the byte to the output stream. Use `BufferedInputStream` and `BufferedOutputStream` to improve the performance. Listing 17.4 gives the solution to the problem.

LISTING 17.4 Copy.java

```
import java.io.*;

public class Copy {
  /** Main method
     @param args[0] for sourcefile
     @param args[1] for target file
   */
  public static void main(String[] args) throws IOException {
    // Check command-line parameter usage
    if (args.length != 2) {
      System.out.println(
        "Usage: java Copy sourceFile targetfile");
      System.exit(1);
    }

    // Check if source file exists
    File sourceFile = new File(args[0]);
    if (!sourceFile.exists()) {
       System.out.println("Source file " + args[0]
         + " does not exist");
       System.exit(2);
    }

    // Check if target file exists
    File targetFile = new File(args[1]);
    if (targetFile.exists()) {
      System.out.println("Target file " + args[1]
        + " already exists");
      System.exit(3);
    }

```

check usage

source file

target file

input stream

output stream

read

write

```
32     try (
33       // Create an input stream
34       BufferedInputStream input =
35         new BufferedInputStream(new FileInputStream(sourceFile));
36 
37       // Create an output stream
38       BufferedOutputStream output =
39         new BufferedOutputStream(new FileOutputStream(targetFile));
40     ) {
41       // Continuously read a byte from input and write it to output
42       int r, numberOfBytesCopied = 0;
43       while ((r = input.read()) != -1) {
44         output.write((byte)r);
45         numberOfBytesCopied++;
46       }
47 
48       // Display the file size
49       System.out.println(numberOfBytesCopied + " bytes copied");
50     }
51   }
52 }
```

The program first checks whether the user has passed the two required arguments from the command line in lines 10–14.

The program uses the `File` class to check whether the source file and target file exist. If the source file does not exist (lines 18–22), or if the target file already exists (lines 25–30), the program ends.

An input stream is created using `BufferedInputStream` wrapped on `FileInputStream` in lines 34–35, and an output stream is created using `BufferedOutputStream` wrapped on `FileOutputStream` in lines 38–39.

The expression `((r = input.read()) != -1)` (line 43) reads a byte from `input.read()`, assigns it to `r`, and checks whether it is `-1`. The input value of `-1` signifies the end of a file. The program continuously reads bytes from the input stream and sends them to the output stream until all of the bytes have been read.

Check Point

17.5.1 How does the program check if a file already exists?

17.5.2 How does the program detect the end of the file while reading data?

17.5.3 How does the program count the number of bytes read from the file?

17.6 Object I/O

Key Point

`ObjectInputStream`/`ObjectOutputStream` classes can be used to read/write serializable objects.

VideoNote
Object I/O

`DataInputStream`/`DataOutputStream` enables you to perform I/O for primitive-type values and strings. `ObjectInputStream`/`ObjectOutputStream` enables you to perform I/O for objects in addition to primitive-type values and strings. Since `ObjectInputStream`/`ObjectOutputStream` contains all the functions of `DataInputStream`/`DataOutputStream`, you can replace `DataInputStream`/`DataOutputStream` completely with `ObjectInputStream`/`ObjectOutputStream`.

`ObjectInputStream` extends `InputStream` and implements `ObjectInput` and `ObjectStreamConstants`, as shown in Figure 17.16. `ObjectInput` is a subinterface of `DataInput` (`DataInput` is shown in Figure 17.9). `ObjectStreamConstants` contains the constants to support `ObjectInputStream`/`ObjectOutputStream`.

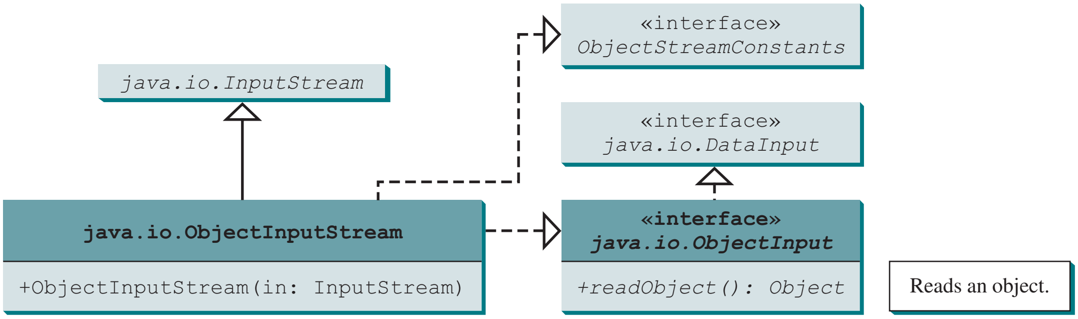

«interface»
ObjectStreamConstants
java.io.InputStream
«interface»
java.io.DataInput
java.io.ObjectInputStream
+ObjectInputStream(in: InputStream)
«interface»
java.io.ObjectInput
+readObject(): Object
Reads an object.

FIGURE 17.16 `ObjectInputStream` can read objects, primitive-type values, and strings.

`ObjectOutputStream` extends `OutputStream` and implements `ObjectOutput` and `ObjectStreamConstants`, as shown in Figure 17.17. `ObjectOutput` is a subinterface of `DataOutput` (`DataOutput` is shown in Figure 17.10).

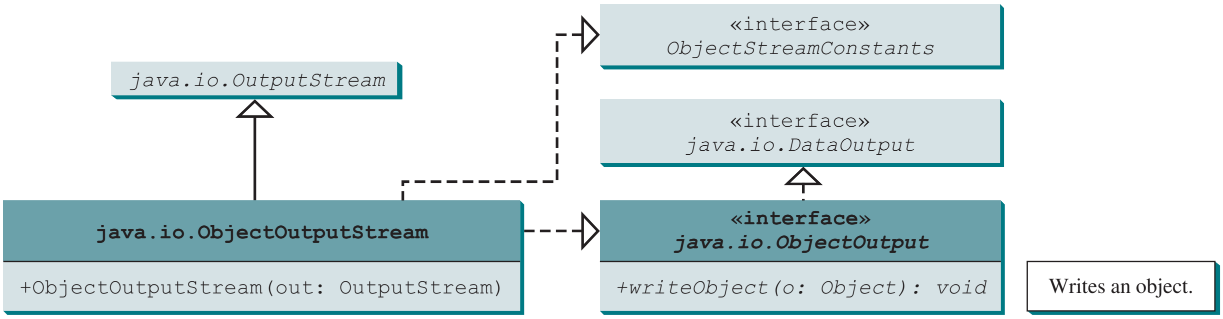

«interface»
ObjectStreamConstants
java.io.OutputStream
«interface»
java.io.DataOutput
java.io.ObjectOutputStream
+ObjectOutputStream(out: OutputStream)
«interface»
java.io.ObjectOutput
+writeObject(o: Object): void
Writes an object.

FIGURE 17.17 `ObjectOutputStream` can write objects, primitive-type values, and strings.

You can wrap an `ObjectInputStream`/`ObjectOutputStream` on any `InputStream`/`OutputStream` using the following constructors:

```
// Create an ObjectInputStream
public ObjectInputStream(InputStream in)

// Create an ObjectOutputStream
public ObjectOutputStream(OutputStream out)
```

Listing 17.5 writes students' names, scores, and the current date to a file named **object.dat**.

LISTING 17.5 TestObjectOutputStream.java

```
import java.io.*;

public class TestObjectOutputStream {
  public static void main(String[] args) throws IOException {
    try ( // Create an output stream for file object.dat
      ObjectOutputStream output =
        new ObjectOutputStream(new FileOutputStream("object.dat"));
    ) {
      // Write a string, double value, and object to the file
      output.writeUTF("John");
      output.writeDouble(85.5);
```

output stream

output string

output object

```
12        output.writeObject(new java.util.Date());
13      }
14    }
15  }
```

An `ObjectOutputStream` is created to write data into the file **object.dat** in lines 6 and 7. A string, a double value, and an object are written to the file in lines 10–12. To improve performance, you may add a buffer in the stream using the following statement to replace lines 6 and 7:

```
ObjectOutputStream output = new ObjectOutputStream(
  new BufferedOutputStream(new FileOutputStream("object.dat")));
```

Multiple objects or primitives can be written to the stream. The objects must be read back from the corresponding `ObjectInputStream` with the same types and in the same order as they were written. Java's safe casting should be used to get the desired type. Listing 17.6 reads data from **object.dat**.

Listing 17.6 TestObjectInputStream.java

input stream

input string

input object

```
 1  import java.io.*;
 2
 3  public class TestObjectInputStream {
 4    public static void main(String[] args)
 5      throws ClassNotFoundException, IOException {
 6      try ( // Create an input stream for file object.dat
 7        ObjectInputStream input =
 8          new ObjectInputStream(new FileInputStream("object.dat"));
 9      ) {
10        // Read a string, double value, and object from the file
11        String name = input.readUTF();
12        double score = input.readDouble();
13        java.util.Date date = (java.util.Date)(input.readObject());
14        System.out.println(name + " " + score + " " + date);
15      }
16    }
17  }
```

```
John 85.5 Sun Dec 04 10:35:31 EST 2011
```

`ClassNotFoundException`

The `readObject()` method may throw `java.lang.ClassNotFoundException` because when the JVM restores an object, it first loads the class for the object if the class has not been loaded. Since `ClassNotFoundException` is a checked exception, the `main` method declares to throw it in line 5. An `ObjectInputStream` is created to read input from **object.dat** in lines 7–8. You have to read the data from the file in the same order and format as they were written to the file. A string, a double value, and an object are read in lines 11–13. Since `readObject()` returns an `Object`, it is cast into `Date` and assigned to a `Date` variable in line 13.

17.6.1 The `Serializable` Interface

serializable

Not every object can be written to an output stream. Objects that can be so written are said to be *serializable*. A serializable object is an instance of the `java.io.Serializable` interface, so the object's class must implement `Serializable`.

The `Serializable` interface is a marker interface. Since it has no methods, you don't need to add additional code in your class that implements `Serializable`. Implementing this interface enables the Java serialization mechanism to automate the process of storing objects and arrays.

To appreciate this automation feature, consider what you otherwise need to do in order to store an object. Suppose that you wish to store an `ArrayList` object. To do this, you need to store all the elements in the list. Each element is an object that may contain other objects. As you can see, this would be a very tedious process. Fortunately, you don't have to go through it manually. Java provides a built-in mechanism to automate the process of writing objects. This process is referred as *object serialization*, which is implemented in `ObjectOutputStream`. In contrast, the process of reading objects is referred as *object deserialization*, which is implemented in `ObjectInputStream`.

serialization

deserialization

Many classes in the Java API implement `Serializable`. All the wrapper classes for primitive-type values: `java.math.BigInteger`, `java.math.BigDecimal`, `java.lang.String`, `java.lang.StringBuilder`, `java.lang.StringBuffer`, `java.util.Date`, and `java.util.ArrayList` implement `java.io.Serializable`. Attempting to store an object that does not support the `Serializable` interface would cause a `NotSerializableException`.

NotSerializableException

When a serializable object is stored, the class of the object is encoded; this includes the class name and the signature of the class, the values of the object's instance variables, and the closure of any other objects referenced by the object. The values of the object's static variables are not stored.

Note

Nonserializable fields

If an object is an instance of `Serializable` but contains nonserializable instance data fields, can it be serialized? The answer is no. To enable the object to be serialized, mark these data fields with the `transient` keyword to tell the JVM to ignore them when writing the object to an object stream. Consider the following class:

```
public class C implements java.io.Serializable {
  private int v1;
  private static double v2;
  private transient A v3 = new A();
}

class A { } // A is not serializable
```

When an object of the `C` class is serialized, only variable `v1` is serialized. Variable `v2` is not serialized because it is a static variable, and variable `v3` is not serialized because it is marked `transient`. If `v3` were not marked `transient`, a `java.io.NotSerializableException` would occur.

transient

Note

Duplicate objects

If an object is written to an object stream more than once, will it be stored in multiple copies? No, it will not. When an object is written for the first time, a serial number is created for it. The JVM writes the complete contents of the object along with the serial number into the object stream. After the first time, only the serial number is stored if the same object is written again. When the objects are read back, their references are the same since only one object is actually created in the memory.

17.6.2 Serializing Arrays

An array is serializable if all its elements are serializable. An entire array can be saved into a file using `writeObject` and later can be restored using `readObject`. Listing 17.7 stores an array of five `int` values and an array of three strings, and reads them back to display on the console.

LISTING 17.7 `TestObjectStreamForArray.java`

output stream

store array

input stream

restore array

```
 1  import java.io.*;
 2
 3  public class TestObjectStreamForArray {
 4    public static void main(String[] args)
 5        throws ClassNotFoundException, IOException {
 6      int[] numbers = {1, 2, 3, 4, 5};
 7      String[] strings = {"John", "Susan", "Kim"};
 8
 9      try ( // Create an output stream for file array.dat
10        ObjectOutputStream output = new ObjectOutputStream(new
11          FileOutputStream("array.dat", true));
12      ) {
13        // Write arrays to the object output stream
14        output.writeObject(numbers);
15        output.writeObject(strings);
16      }
17
18      try ( // Create an input stream for file array.dat
19        ObjectInputStream input =
20          new ObjectInputStream(new FileInputStream("array.dat"));
21      ) {
22        int[] newNumbers = (int[])(input.readObject());
23        String[] newStrings = (String[])(input.readObject());
24
25        // Display arrays
26        for (int i = 0; i < newNumbers.length; i++)
27          System.out.print(newNumbers[i] + " ");
28        System.out.println();
29
30        for (int i = 0; i < newStrings.length; i++)
31          System.out.print(newStrings[i] + " ");
32      }
33    }
34  }
```

```
1 2 3 4 5
John Susan Kim
```

Lines 14–15 write two arrays into file **array.dat**. Lines 22–23 read two arrays back in the same order they were written. Since `readObject()` returns `Object`, casting is used to cast the objects into `int[]` and `String[]`.

Check Point

17.6.1 Is it true that `DataInputStream`/`DataOutputStream` can always be replaced by `ObjectInputStream`/`ObjectOutputStream`?

17.6.2 What types of objects can be stored using the `ObjectOutputStream`? What is the method for writing an object? What is the method for reading an object? What is the return type of the method that reads an object from `ObjectInputStream`?

17.6.3 If you serialize two objects of the same type, will they take the same amount of space? If not, give an example.

17.6.4 Is it true that any instance of `java.io.Serializable` can be successfully serialized? Are the static variables in an object serialized? How do you mark an instance variable not to be serialized?

17.6.5 What will happen when you attempt to run the following code?

```
import java.io.*;

public class Test {
  public static void main(String[] args) throws IOException {
    try ( ObjectOutputStream output =
        new ObjectOutputStream(new FileOutputStream("object.dat")); ) {
      output.writeObject(new A());
    }
  }
}

class A implements Serializable {
  B b = new B();
}

class B {
}
```

17.6.6 Can you write an array to an `ObjectOutputStream`?

17.7 Random-Access Files

Key Point

Java provides the `RandomAccessFile` *class to allow data to be read from and written to at any locations in the file.*

All of the streams you have used so far are known as *read-only* or *write-only* streams. These streams are called *sequential streams*. A file that is opened using a sequential stream is called a *sequential-access file*. The contents of a sequential-access file cannot be updated. However, it is often necessary to modify files. Java provides the `RandomAccessFile` class to allow data to be read from and written to at any locations in the file. A file that is opened using the `RandomAccessFile` class is known as a *random-access file*.

read-only
write-only
sequential-access file

random-access file

The `RandomAccessFile` class implements the `DataInput` and `DataOutput` interfaces, as shown in Figure 17.18. The `DataInput` interface (see Figure 17.9) defines the methods for reading primitive-type values and strings (e.g., `readInt`, `readDouble`, `readChar`, `readBoolean`, and `readUTF`) and the `DataOutput` interface (see Figure 17.10) defines the methods for writing primitive-type values and strings (e.g., `writeInt`, `writeDouble`, `writeChar`, `writeBoolean`, and `writeUTF`).

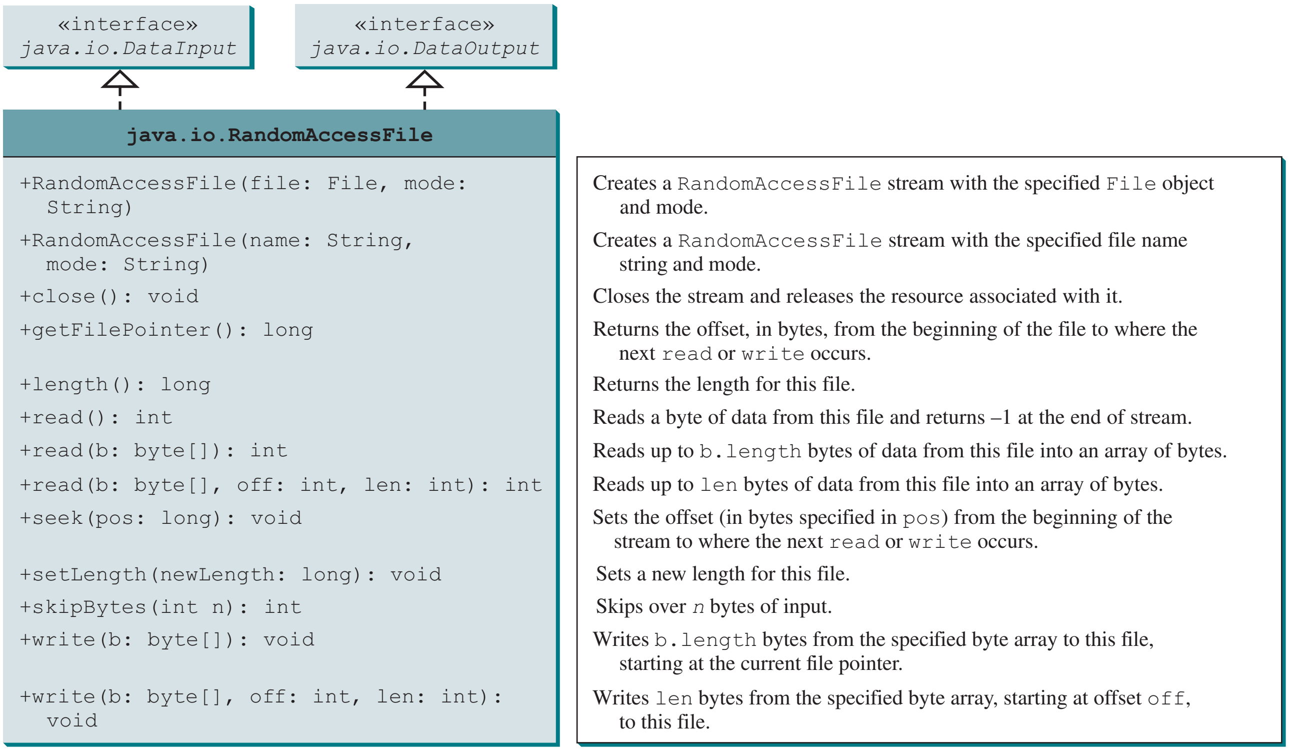
When creating a `RandomAccessFile`, you can specify one of the two modes: `r` or `rw`. Mode `r` means that the stream is read-only, and mode `rw` indicates that the stream allows both read and write. For example, the following statement creates a new stream, `raf`, that allows the program to read from and write to the file **test.dat**:

```
RandomAccessFile raf = new RandomAccessFile("test.dat", "rw");
```

If **test.dat** already exists, `raf` is created to access it; if **test.dat** does not exist, a new file named **test.dat** is created and `raf` is created to access the new file. The method `raf.length()` returns the number of bytes in **test.dat** at any given time. If you append new data into the file, `raf.length()` increases.

Tip

If the file is not intended to be modified, open it with the `r` mode. This prevents unintentional modification of the file.


«interface»
java.io.DataInput
«interface»
java.io.DataOutput
java.io.RandomAccessFile

`+RandomAccessFile(file: File, mode: String)`	Creates a `RandomAccessFile` stream with the specified `File` object and mode.
`+RandomAccessFile(name: String, mode: String)`	Creates a `RandomAccessFile` stream with the specified file name string and mode.
`+close(): void`	Closes the stream and releases the resource associated with it.
`+getFilePointer(): long`	Returns the offset, in bytes, from the beginning of the file to where the next `read` or `write` occurs.
`+length(): long`	Returns the length for this file.
`+read(): int`	Reads a byte of data from this file and returns –1 at the end of stream.
`+read(b: byte[]): int`	Reads up to `b.length` bytes of data from this file into an array of bytes.
`+read(b: byte[], off: int, len: int): int`	Reads up to `len` bytes of data from this file into an array of bytes.
`+seek(pos: long): void`	Sets the offset (in bytes specified in `pos`) from the beginning of the stream to where the next `read` or `write` occurs.
`+setLength(newLength: long): void`	Sets a new length for this file.
`+skipBytes(int n): int`	Skips over *n* bytes of input.
`+write(b: byte[]): void`	Writes `b.length` bytes from the specified byte array to this file, starting at the current file pointer.
`+write(b: byte[], off: int, len: int): void`	Writes `len` bytes from the specified byte array, starting at offset `off`, to this file.

FIGURE 17.18 **`RandomAccessFile`** implements the **`DataInput`** and **`DataOutput`** interfaces with additional methods to support random access.

file pointer

A random-access file consists of a sequence of bytes. A special marker called a *file pointer* is positioned at one of these bytes. A read or write operation takes place at the location of the file pointer. When a file is opened, the file pointer is set at the beginning of the file. When you read from or write data to the file, the file pointer moves forward to the next data item. For example, if you read an **`int`** value using **`readInt()`**, the JVM reads **`4`** bytes from the file pointer and now the file pointer is **`4`** bytes ahead of the previous location, as shown in Figure 17.19.

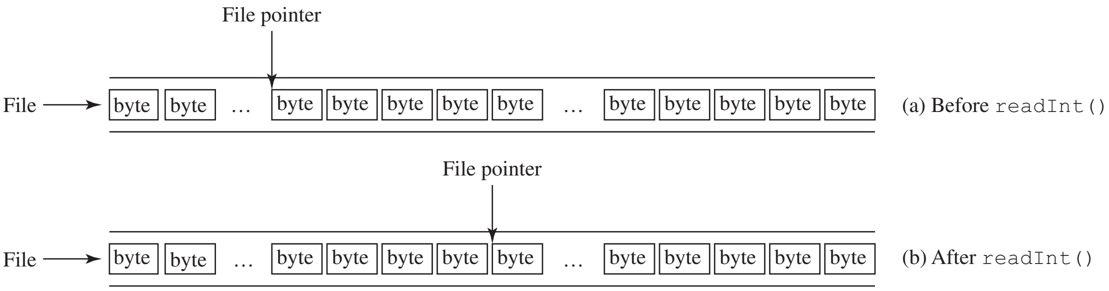

File pointer
File
byte byte … byte byte byte byte byte … byte byte byte byte byte
(a) Before readInt()
File pointer
File
byte byte … byte byte byte byte byte … byte byte byte byte byte
(b) After readInt()

FIGURE 17.19 After an **`int`** value is read, the file pointer is moved 4 bytes ahead.

For a **`RandomAccessFile raf`**, you can use the **`raf.seek(position)`** method to move the file pointer to a specified position. **`raf.seek(0)`** moves it to the beginning of the file and **`raf.seek(raf.length())`** moves it to the end of the file. Listing 17.8 demonstrates **`RandomAccessFile`**. A large case study of using **`RandomAccessFile`** to organize an address book is given in Supplement VI.D.

LISTING 17.8 TestRandomAccessFile.java

```
 1  import java.io.*;
 2
 3  public class TestRandomAccessFile {
 4    public static void main(String[] args) throws IOException {
 5      try ( // Create a random access file
 6        RandomAccessFile inout = new RandomAccessFile("inout.dat", "rw");
 7      ) {
 8        // Clear the file to destroy the old contents if exists
 9        inout.setLength(0);
10
11        // Write new integers to the file
12        for (int i = 0; i < 200; i++)
13          inout.writeInt(i);
14
15        // Display the current length of the file
16        System.out.println("Current file length is " + inout.length());
17
18        // Retrieve the first number
19        inout.seek(0); // Move the file pointer to the beginning
20        System.out.println("The first number is " + inout.readInt());
21
22        // Retrieve the second number
23        inout.seek(1 * 4); // Move the file pointer to the second number
24        System.out.println("The second number is " + inout.readInt());
25
26        // Retrieve the tenth number
27        inout.seek(9 * 4); // Move the file pointer to the tenth number
28        System.out.println("The tenth number is " + inout.readInt());
29
30        // Modify the eleventh number
31        inout.writeInt(555);
32
33        // Append a new number
34        inout.seek(inout.length()); // Move the file pointer to the end
35        inout.writeInt(999);
36
37        // Display the new length
38        System.out.println("The new length is " + inout.length());
39
40        // Retrieve the new eleventh number
41        inout.seek(10 * 4); // Move the file pointer to the eleventh number
42        System.out.println("The eleventh number is " + inout.readInt());
43      }
44    }
45  }
```

`RandomAccessFile`

empty file

write

move pointer

read

```
Current file length is 800
The first number is 0
The second number is 1
The tenth number is 9
The new length is 804
The eleventh number is 555
```

A `RandomAccessFile` is created for the file named **inout.dat** with mode `rw` to allow both read and write operations in line 6.

`inout.setLength(0)` sets the length to `0` in line 9. This, in effect, deletes the old contents of the file.

The `for` loop writes `200` `int` values from `0` to `199` into the file in lines 12–13. Since each `int` value takes `4` bytes, the total length of the file returned from `inout.length()` is now `800` (line 16), as shown in the sample output.

Invoking `inout.seek(0)` in line 19 sets the file pointer to the beginning of the file. `inout.readInt()` reads the first value in line 20 and moves the file pointer to the next number. The second number is read in line 24.

`inout.seek(9 * 4)` (line 27) moves the file pointer to the tenth number. `inout.readInt()` reads the tenth number and moves the file pointer to the eleventh number in line 28. `inout.write(555)` writes a new eleventh number at the current position (line 31). The previous eleventh number is deleted.

`inout.seek(inout.length())` moves the file pointer to the end of the file (line 34). `inout.writeInt(999)` writes a `999` to the file (line 35). Now the length of the file is increased by `4`, so `inout.length()` returns `804` (line 38).

`inout.seek(10 * 4)` moves the file pointer to the eleventh number in line 41. The new eleventh number, `555`, is displayed in line 42.

Check Point

17.7.1 Can `RandomAccessFile` streams read and write a data file created by `DataOutputStream`? Can `RandomAccessFile` streams read and write objects?

17.7.2 Create a `RandomAccessFile` stream for the file **address.dat** to allow the updating of student information in the file. Create a `DataOutputStream` for the file **address.dat**. Explain the differences between these two statements.

17.7.3 What happens if the file **test.dat** does not exist when you attempt to compile and run the following code?

```java
import java.io.*;

public class Test {
  public static void main(String[] args) {
    try ( RandomAccessFile raf =
        new RandomAccessFile("test.dat", "r"); ) {
      int i = raf.readInt();
    }
    catch (IOException ex) {
      System.out.println("IO exception");
    }
  }
}
```

KEY TERMS

binary I/O	692
deserialization	709
file pointer	712
random-access file	711
sequential-access file	711
serialization	709
stream	692
text I/O	692

CHAPTER SUMMARY

1. I/O can be classified into *text I/O* and *binary I/O*. Text I/O interprets data in sequences of characters. Binary I/O interprets data as raw binary values. How text is stored in a file depends on the encoding scheme for the file. Java automatically performs encoding and decoding for text I/O.
2. The `InputStream` and `OutputStream` classes are the roots of all binary I/O classes. `FileInputStream/FileOutputStream` associates a file for input/output. `BufferedInputStream`/`BufferedOutputStream` can be used to wrap any binary I/O stream to improve performance. `DataInputStream`/`DataOutputStream` can be used to read/write primitive values and strings.
3. `ObjectInputStream`/`ObjectOutputStream` can be used to read/write objects in addition to primitive values and strings. To enable object *serialization*, the object's defining class must implement the `java.io.Serializable` marker interface.
4. The `RandomAccessFile` class enables you to read and write data to a file. You can open a file with the `r` mode to indicate that it is read-only, or with the `rw` mode to indicate that it is updateable. Since the `RandomAccessFile` class implements `DataInput` and `DataOutput` interfaces, many methods in `RandomAccessFile` are the same as those in `DataInputStream` and `DataOutputStream`.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

Section 17.3

***17.1** (*Create a text file*) Write a program to create a file named **Exercise17_01.txt** if it does not exist. Append new data to it if it already exists. Write 100 integers created randomly into the file using text I/O. Integers are separated by a space.

Section 17.4

***17.2** (*Create a binary data file*) Write a program to create a file named **Exercise17_02.dat** if it does not exist. Append new data to it if it already exists. Write 100 integers created randomly into the file using binary I/O.

***17.3** (*Sum all the integers in a binary data file*) Suppose a binary data file named **Exercise17_02.dat** has been created from Programming Exercise 17.2 and its data are created using `writeInt(int)` in `DataOutputStream`. The file contains an unspecified number of integers. Write a program to find the sum of the integers.

***17.4** (*Convert a text file into UTF*) Write a program that reads lines of characters from a text file and writes each line as a UTF string into a binary file. Display the sizes of the text file and the binary file. Use the following command to run the program:

```
java Exercise17_04 Welcome.java Welcome.utf
```

Section 17.6

*17.5 (*Store objects and arrays in a file*) Write a program that stores an array of the five `int` values `1`, `2`, `3`, `4`, and `5`, a `Date` object for the current time, and the `double` value `5.5` into the file named **Exercise17_05.dat**. In the same program, write the code to read and display the data.

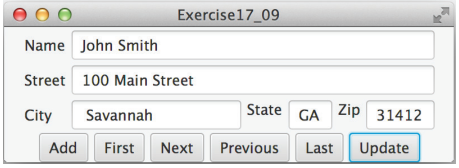
*17.6 (*Store `Loan` objects*) The `Loan` class in Listing 10.2 does not implement `Serializable`. Rewrite the `Loan` class to implement `Serializable`. Write a program that creates five `Loan` objects and stores them in a file named **Exercise17_06.dat**.

*17.7 (*Restore objects from a file*) Suppose a file named **Exercise17_06.dat** has been created using the `ObjectOutputStream` from the preceding programming exercises. The file contains `Loan` objects. The `Loan` class in Listing 10.2 does not implement `Serializable`. Rewrite the `Loan` class to implement `Serializable`. Write a program that reads the `Loan` objects from the file and displays the total loan amount. Suppose that you don't know how many `Loan` objects are there in the file, use `EOFException` to end the loop.

Section 17.7

*17.8 (*Update count*) Suppose that you wish to track how many times a program has been executed. You can store an `int` to count the file. Increase the count by `1` each time this program is executed. Let the program be **Exercise17_08.txt** and store the count in **Exercise17_08.dat**.

***17.9 (*Address book*) Write a program that stores, retrieves, adds, and updates addresses as shown in Figure 17.20. Use a fixed-length string for storing each attribute in the address. Use random-access file for reading and writing an address. Assume the sizes of the name, street, city, state, and zip are 32, 32, 20, 2, and 5 bytes, respectively.



Name: John Smith
Street: 100 Main Street
City: Savannah
State: GA
Zip: 31412

FIGURE 17.20 The application can store, retrieve, and update addresses from a file. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Comprehensive

VideoNote
Split a large file

*17.10 (*Split files*) Suppose you want to back up a huge file (e.g., a 10-GB AVI file) to a CD-R. You can achieve it by splitting the file into smaller pieces and backing up these pieces separately. Write a utility program that splits a large file into smaller ones using the following command:

```
java Exercise17_10 SourceFile numberOfPieces
```

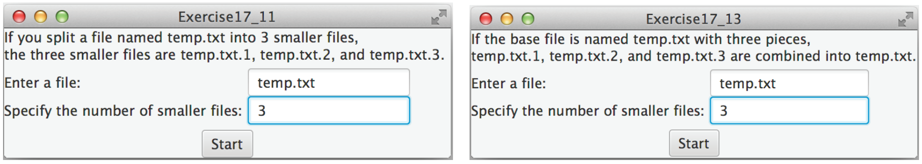
The command creates the files **SourceFile.1**, **SourceFile.2**, . . . , **SourceFile.n**, where **n** is `numberOfPieces` and the output files are about the same size.

**17.11 (*Split files GUI*) Rewrite Exercise 17.10 with a GUI, as shown in Figure 17.21a.

*17.12 (*Combine files*) Write a utility program that combines the files together into a new file using the following command:

```
java Exercise17_12 SourceFile1 . . . SourceFilen TargetFile
```

The command combines SourceFile1, . . . , and SourceFilen into TargetFile.



If you split a file named temp.txt into 3 smaller files,
the three smaller files are temp.txt.1, temp.txt.2, and temp.txt.3.

Enter a file: temp.txt

Specify the number of smaller files: 3

If the base file is named temp.txt with three pieces,
temp.txt.1, temp.txt.2, and temp.txt.3 are combined into temp.txt.

Enter a file: temp.txt

Specify the number of smaller files: 3

(a) (b)

FIGURE 17.21 (a) The program splits a file. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) The program combines files into a new file.

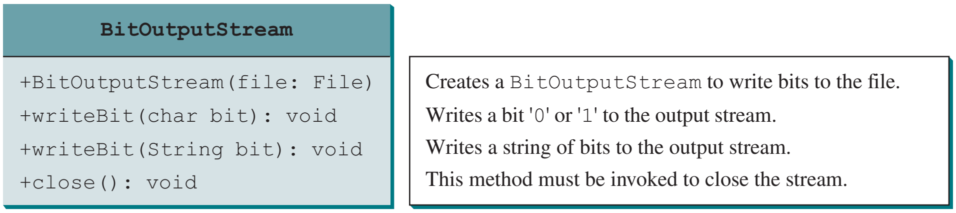
***17.13** (*Combine files GUI*) Rewrite Exercise 17.12 with a GUI, as shown in Figure 17.21b.

17.14 (*Encrypt files*) Encode the file by adding `5` to every byte in the file. Write a program that prompts the user to enter an input file name and an output file name and saves the encrypted version of the input file to the output file.

17.15 (*Decrypt files*) Suppose a file is encrypted using the scheme in Programming Exercise 17.14. Write a program to decode an encrypted file. Your program should prompt the user to enter an input file name for the encrypted file and an output file name for the unencrypted version of the input file.

17.16 (*Frequency of characters*) Write a program that prompts the user to enter the name of an ASCII text file and displays the frequency of the characters in the file.

****17.17** (*`BitOutputStream`*) Implement a class named `BitOutputStream`, as shown in Figure 17.22, for writing bits to an output stream. The `writeBit(char bit)` method stores the bit in a byte variable. When you create a `BitOutputStream`, the byte is empty. After invoking `writeBit('1')`, the byte becomes `00000001`. After invoking `writeBit("0101")`, the byte becomes `00010101`. The first three bits are not filled yet. When a byte is full, it is sent to the output stream. Now the byte is reset to empty. You must close the stream by invoking the `close()` method. If the byte is neither empty nor full, the `close()` method first fills the zeros to make a full `8` bits in the byte and then outputs the byte and closes the stream. For a hint, see Programming Exercise 5.44. Write a test program that sends the bits `010000100100001001101` to the file named **Exercise17_17.dat**.



BitOutputStream	
`+BitOutputStream(file: File)`	Creates a `BitOutputStream` to write bits to the file.
`+writeBit(char bit): void`	Writes a bit '`0`' or '`1`' to the output stream.
`+writeBit(String bit): void`	Writes a string of bits to the output stream.
`+close(): void`	This method must be invoked to close the stream.

FIGURE 17.22 `BitOutputStream` outputs a stream of bits to a file.

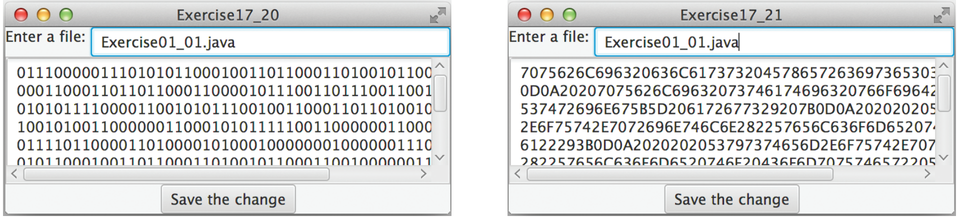
***17.18** (*View bits*) Write the following method that displays the bit representation for the last byte in an integer:

```
public static String getBits(int value)
```

For a hint, see Programming Exercise 5.44. Write a program that prompts the user to enter a file name, reads bytes from the file, and displays each byte's binary representation.

***17.19** (*View hex*) Write a program that prompts the user to enter a file name, reads bytes from the file, and displays each byte's hex representation. (*Hint*: You can first convert the byte value into an 8-bit string, then convert the bit string into a two-digit hex string.)

****17.20** (*Binary editor*) Write a GUI application that lets the user to enter a file name in the text field and press the *Enter* key to display its binary representation in a text area. The user can also modify the binary code and save it back to the file, as shown in Figure 17.23a.



Exercise01_01.java

```
0111000001110101011000100110110001101001011100
0001100011011011000110000101110011011100110011001
0101011110000110010101110010011000110110100101
1001010011000000110001010111110011000000110000
0111101100001101000010100010000000100000001110
01011000100110110001101001011000110010000000011
```

Exercise01_01.java

```
7075626C696320636C617373204578657263697365303
0D0A20207075626C69632073746174696320766F69642
537472696E675B5D20617267732907B0D0A2020202050
2E6F75742E7072696E746C6E282257656C636F6D652074
6122293B0D0A2020202053797374656D2E6F75742E707
282257656C636F6D6520746F20436F6D7075746572205
```

(a) (b)

FIGURE 17.23 The programs enable the user to manipulate the contents of the file in (a) binary (b) hex. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****17.21** (*Hex editor*) Write a GUI application that lets the user to enter a file name in the text field and press the *Enter* key to display its hex representation in a text area. The user can also modify the hex code and save it back to the file, as shown in Figure 17.23b.

CHAPTER

18

RECURSION

Objectives

- To describe what a recursive method is and the benefits of using recursion (§18.1).
- To develop recursive methods for recursive mathematical functions (§§18.2 and 18.3).
- To explain how recursive method calls are handled in a call stack (§§18.2 and 18.3).
- To solve problems using recursion (§18.4).
- To use an overloaded helper method to design a recursive method (§18.5).
- To implement a selection sort using recursion (§18.5.1).
- To implement a binary search using recursion (§18.5.2).
- To get the directory size using recursion (§18.6).
- To solve the Tower of Hanoi problem using recursion (§18.7).
- To draw fractals using recursion (§18.8).
- To discover the relationship and difference between recursion and iteration (§18.9).
- To know tail-recursive methods and why they are desirable (§18.10).

18.1 Introduction

Key Point

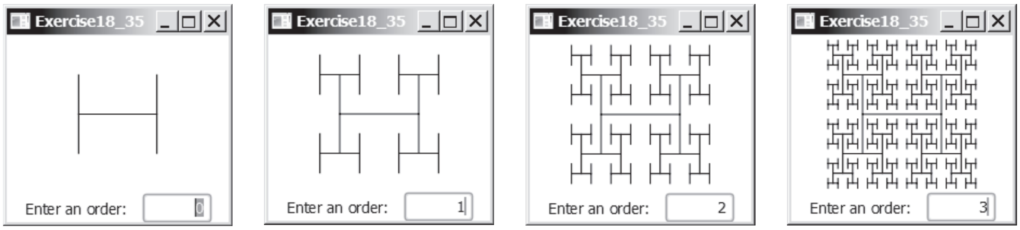
Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops.

search word problem

H-tree problem

Suppose you want to find all the files under a directory that contains a particular word. How do you solve this problem? There are several ways to do so. An intuitive and effective solution is to use recursion by searching the files in the subdirectories recursively.

H-trees, depicted in Figure 18.1, are used in a very large-scale integration (VLSI) design as a clock distribution network for routing timing signals to all parts of a chip with equal propagation delays. How do you write a program to display H-trees? A good approach is to use recursion.



Enter an order: 0

Enter an order: 1

Enter an order: 2

Enter an order: 3

(a) (b) (c) (d)

Figure 18.1 An H-tree can be displayed using recursion. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

recursive method

To use recursion is to program using *recursive methods*—that is, to use methods that invoke themselves. Recursion is a useful programming technique. In some cases, it enables you to develop a natural, straightforward, simple solution to an otherwise difficult problem. This chapter introduces the concepts and techniques of recursive programming and illustrates with examples of how to "think recursively."

18.2 Case Study: Computing Factorials

Key Point

A recursive method is one that invokes itself directly or indirectly.

Many mathematical functions are defined using recursion. Let's begin with a simple example. The factorial of a number `n` can be recursively defined as follows:

```
0! = 1;
n! = n × (n − 1)!; n > 0
```

How do you find `n!` for a given `n`? To find `1!` is easy because you know that `0!` is `1` and `1!` is `1` × `0!`. Assuming that you know `(n - 1)!`, you can obtain `n!` immediately by using `n` × `(n - 1)!`. Thus, the problem of computing `n!` is reduced to computing `(n - 1)!`. When computing `(n - 1)!`, you can apply the same idea recursively until `n` is reduced to `0`.

base case or stopping condition

recursive call

Let `factorial(n)` be the method for computing `n!`. If you call the method with `n = 0`, it immediately returns the result. The method knows how to solve the simplest case, which is referred to as the *base case* or the *stopping condition*. If you call the method with `n > 0`, it reduces the problem into a subproblem for computing the factorial of `n - 1`. The *subproblem* is essentially the same as the original problem, but it is simpler or smaller. Because the subproblem has the same property as the original problem, you can call the method with a different argument, which is referred to as a *recursive call*.

The recursive algorithm for computing `factorial(n)` can be simply described as follows:

```
if (n == 0)
  return 1;
  else
    return n * factorial(n - 1);
```

A recursive call can result in many more recursive calls because the method keeps on dividing a subproblem into new subproblems. For a recursive method to terminate, the problem must eventually be reduced to a stopping case, at which point the method returns a result to its caller. The caller then performs a computation and returns the result to its own caller. This process continues until the result is passed back to the original caller. The original problem can now be solved by multiplying `n` by the result of `factorial(n - 1)`.

Listing 18.1 gives a complete program that prompts the user to enter a nonnegative integer and displays the factorial for the number.

Listing 18.1 ComputeFactorial.java

```
 1  import java.util.Scanner;
 2
 3  public class ComputeFactorial {
 4    /** Main method */
 5    public static void main(String[] args) {
 6      // Create a Scanner
 7      Scanner input = new Scanner(System.in);
 8      System.out.print("Enter a nonnegative integer: ");
 9      int n = input.nextInt();
10
11      // Display factorial
12      System.out.println("Factorial of " + n + " is " + factorial(n));
13    }
14
15    /** Return the factorial for the specified number */
16    public static long factorial(int n) {
17      if (n == 0) // Base case
18        return 1;
19      else
20        return n *  factorial(n - 1); // Recursive call
21    }
22  }
```

base case

recursion

```
Enter a nonnegative integer: 4 ↵Enter
Factorial of 4 is 24
```

```
Enter a nonnegative integer: 10 ↵Enter
Factorial of 10 is 3628800
```

The `factorial` method (lines 16–21) is essentially a direct translation of the recursive mathematical definition for the factorial into Java code. The call to `factorial` is recursive because it calls itself. The parameter passed to `factorial` is decremented until it reaches the base case of `0`.

how does it work?

You see how to write a recursive method. How does recursion work behind the scenes? Figure 18.2 illustrates the execution of the recursive calls, starting with `n = 4`. The use of stack space for recursive calls is shown in Figure 18.3.

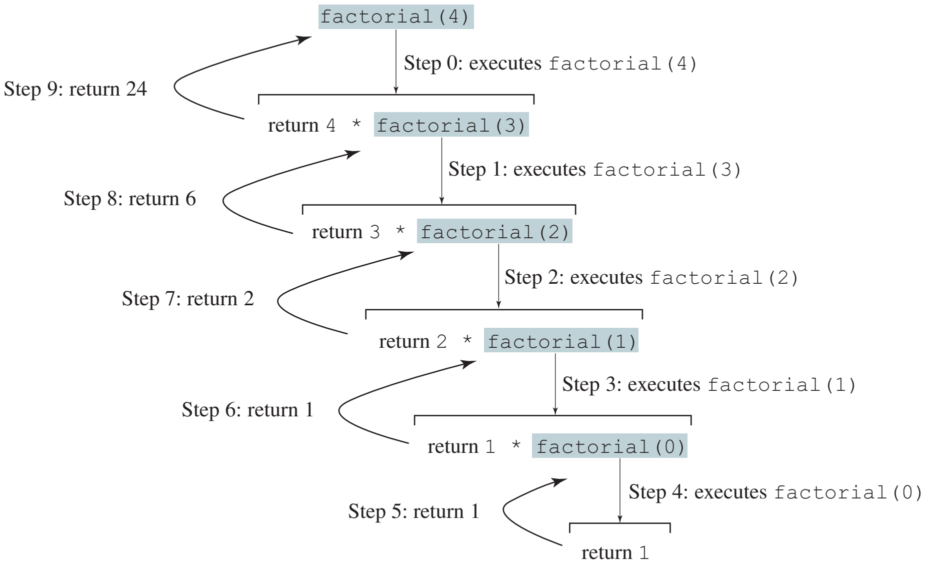

factorial(4)
Step 0: executes factorial(4)
Step 9: return 24
return 4 * factorial(3)
Step 1: executes factorial(3)
Step 8: return 6
return 3 * factorial(2)
Step 2: executes factorial(2)
Step 7: return 2
return 2 * factorial(1)
Step 3: executes factorial(1)
Step 6: return 1
return 1 * factorial(0)
Step 4: executes factorial(0)
Step 5: return 1
return 1

FIGURE 18.2 Invoking `factorial(4)` spawns recursive calls to `factorial`.

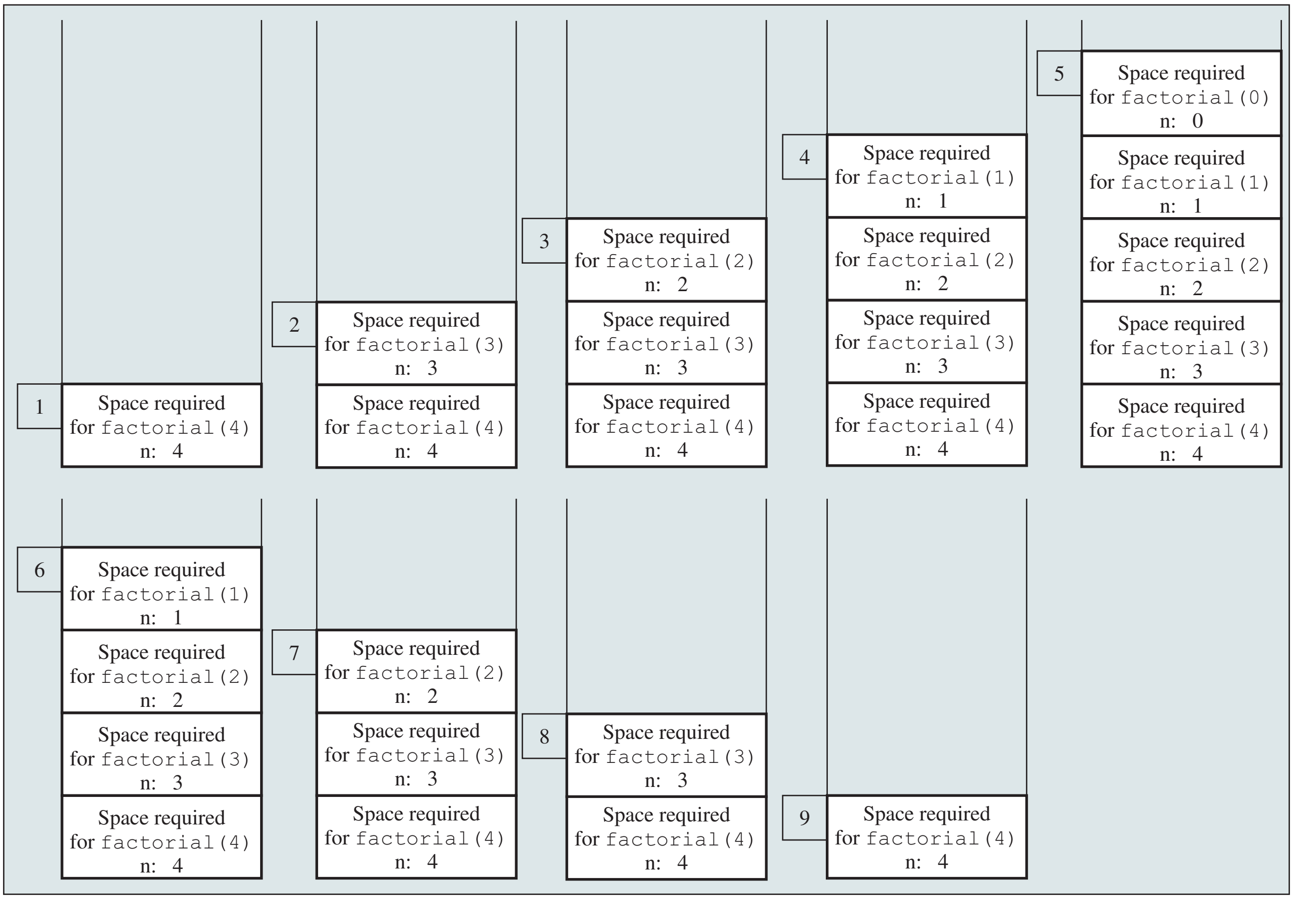

1
Space required for factorial(4) n: 4
2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
3
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
4
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
5
Space required for factorial(0) n: 0
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
6
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
7
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
8
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
9
Space required for factorial(4) n: 4

FIGURE 18.3 When `factorial(4)` is being executed, the `factorial` method is called recursively, causing the stack space to dynamically change.

Pedagogical Note

It is simpler and more efficient to implement the `factorial` method using a loop. However, we use the recursive `factorial` method here to demonstrate the concept of recursion. Later in this chapter, we will present some problems that are inherently recursive and are difficult to solve without using recursion.

Note

If recursion does not reduce the problem in a manner that allows it to eventually converge into the base case or a base case is not specified, *infinite recursion* can occur. For example, suppose you mistakenly write the `factorial` method as follows:

infinite recursion

```
public static long factorial(int n) {
  return n * factorial(n - 1);
}
```

The method runs infinitely and causes a `StackOverflowError`.

The example discussed in this section shows a recursive method that invokes itself. This is known as *direct recursion*. It is also possible to create *indirect recursion*. This occurs when method `A` invokes method `B`, which in turn directly or indirectly invokes method `A`.

direct recursion

indirect recursion

Check Point

18.2.1 What is a recursive method? What is an infinite recursion?

18.2.2 How many times is the `factorial` method in Listing 18.1 invoked for `factorial(6)`?

18.2.3 Show the output of the following programs and identify base cases and recursive calls.

```
public class Test {
  public static void main(String[] args) {
    System.out.println(
      "Sum is " + xMethod(5));
  }

  public static int xMethod(int n) {
    if (n == 1)
      return 1;
    else
      return n + xMethod(n - 1);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    xMethod(1234567);
  }

  public static void xMethod(int n) {
    if (n > 0) {
      System.out.print(n % 10);
      xMethod(n / 10);
    }
  }
}
```

18.2.4 Write a recursive mathematical definition for computing 2^n for a positive integer n.

18.2.5 Write a recursive mathematical definition for computing x^n for a positive integer n and a real number x.

18.2.6 Write a recursive mathematical definition for computing $1 + 2 + 3 + \cdots + n$ for a positive integer n.

18.3 Case Study: Computing Fibonacci Numbers

In some cases, recursion enables you to create an intuitive, straightforward, simple solution to a problem.

Key Point

The `factorial` method in the preceding section could easily be rewritten without using recursion. In this section, we show an example for creating an intuitive solution to a problem using recursion. Consider the well-known Fibonacci-series problem:

The series:	0	1	1	2	3	5	8	13	21	34	55	89	. . .
indexes:	0	1	2	3	4	5	6	7	8	9	10	11	

The Fibonacci series begins with `0` and `1`, and each subsequent number is the sum of the preceding two. The series can be recursively defined as

```
fib(0) = 0;
fib(1) = 1;
fib(index) = fib(index - 2) + fib(index - 1); index >= 2
```

The Fibonacci series was named for Leonardo Fibonacci, a medieval mathematician, who originated it to model the growth of the rabbit population. It can be applied in numeric optimization and in various other areas.

How do you find `fib(index)` for a given `index`? It is easy to find `fib(2)` because you know `fib(0)` and `fib(1)`. Assuming you know `fib(index - 2)` and `fib(index - 1)`, you can obtain `fib(index)` immediately. Thus, the problem of computing `fib(index)` is reduced to computing `fib(index - 2)` and `fib(index - 1)`. When doing so, you apply the idea recursively until `index` is reduced to `0` or `1`.

The base case is `index = 0` or `index = 1`. If you call the method with `index = 0` or `index = 1`, it immediately returns the result. If you call the method with `index >= 2`, it divides the problem into two subproblems for computing `fib(index - 1)` and `fib(index - 2)` using recursive calls. The recursive algorithm for computing `fib(index)` can be simply described as follows:

```
if (index == 0)
  return 0;
else if (index == 1)
  return 1;
else
  return fib(index - 1) + fib(index - 2);
```

Listing 18.2 gives a complete program that prompts the user to enter an index and computes the Fibonacci number for that index.

LISTING 18.2 ComputeFibonacci.java

```
import java.util.Scanner;

public class ComputeFibonacci {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an index for a Fibonacci number: ");
    int index = input.nextInt();

    // Find and display the Fibonacci number
    System.out.println("The Fibonacci number at index "
      + index + " is " + fib(index));
  }

  /** The method for finding the Fibonacci number */
  public static long fib(long index) {
    if (index == 0) // Base case
      return 0;
    else if (index == 1) // Base case
      return 1;
    else  // Reduction and recursive calls
      return fib(index - 1) + fib(index - 2);
  }
}
```

base case

base case

recursion

```
Enter an index for a Fibonacci number: 1 ↵Enter
The Fibonacci number at index 1 is 1
```

```
Enter an index for a Fibonacci number: 6 ↵Enter
The Fibonacci number at index 6 is 8
```

```
Enter an index for a Fibonacci number: 7 ↵Enter
The Fibonacci number at index 7 is 13
```

The program does not show the considerable amount of work done behind the scenes by the computer. Figure 18.4, however, shows the successive recursive calls for evaluating `fib(4)`. The original method, `fib(4)`, makes two recursive calls, `fib(3)` and `fib(2)`, and then returns `fib(3) + fib(2)`. However, in what order are these methods called? In Java, operands are evaluated from left to right, so `fib(2)` is called after `fib(3)` is completely evaluated. The labels in Figure 18.4 show the order in which the methods are called.

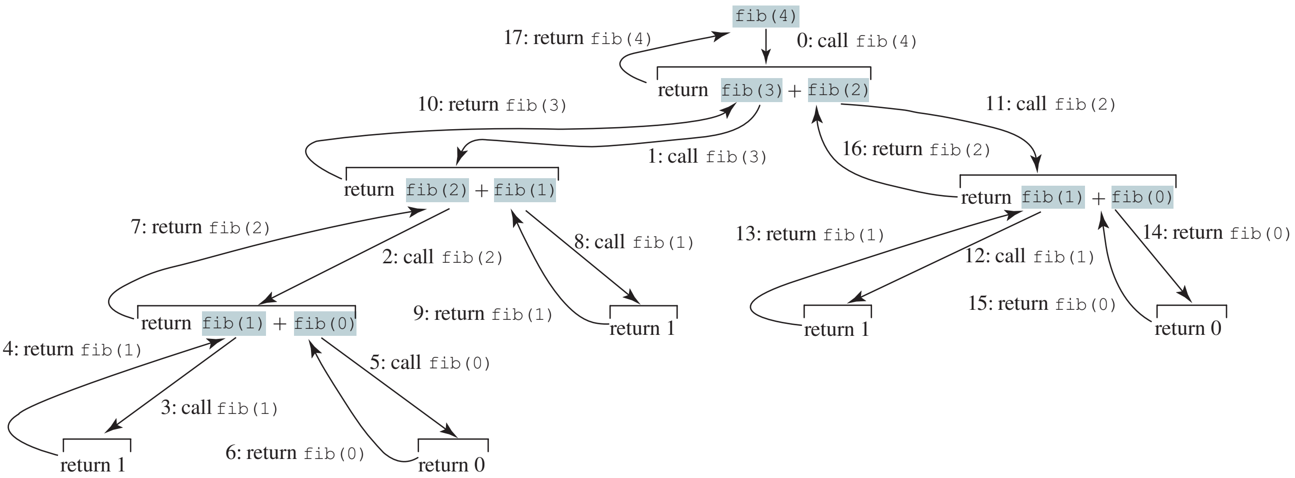

fib(4)
17: return fib(4)
0: call fib(4)
return fib(3) + fib(2)
10: return fib(3)
11: call fib(2)
1: call fib(3)
16: return fib(2)
return fib(2) + fib(1)
return fib(1) + fib(0)
7: return fib(2)
8: call fib(1)
13: return fib(1)
14: return fib(0)
2: call fib(2)
12: call fib(1)
9: return fib(1)
15: return fib(0)
return fib(1) + fib(0)
return 1
return 1
return 0
4: return fib(1)
5: call fib(0)
3: call fib(1)
6: return fib(0)
return 1
return 0

Figure 18.4 Invoking `fib(4)` spawns recursive calls to `fib`.

As shown in Figure 18.4, there are many duplicated recursive calls. For instance, `fib(2)` is called twice, `fib(1)` three times, and `fib(0)` twice. In general, computing `fib(index)` requires roughly twice as many recursive calls as does computing `fib(index - 1)`. As you try larger index values, the number of calls substantially increases, as given in Table 18.1.

Table 18.1 Number of Recursive Calls in `fib(index)`

`index`	2	3	4	10	20	30	40	50
# of calls	3	5	9	177	21,891	2,692,537	331,160,281	2,075,316,483

Pedagogical Note

The recursive implementation of the `fib` method is very simple and straightforward, but it isn't efficient, because it requires more time and memory to run recursive methods. See Programming Exercise 18.2 for an efficient solution using loops. Though it is not practical, the recursive `fib` method is a good example of how to write recursive methods.

Check Point

18.3.1 Show the output of the following two programs:

```
public class Test {
  public static void main(String[] args) {
    xMethod(5);
  }

  public static void xMethod(int n) {
    if (n > 0) {
      System.out.print(n + " ");
      xMethod(n - 1);
    }
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    xMethod(5);
  }

  public static void xMethod(int n) {
    if (n > 0) {
      xMethod(n - 1);
      System.out.print(n + " ");
    }
  }
}
```

18.3.2 What is wrong in the following methods?

```
public class Test {
  public static void main(String[] args) {
    xMethod(1234567);
  }

  public static void xMethod(double n) {
    if (n != 0) {
      System.out.print(n);
      xMethod(n / 10);
    }
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Test test = new Test();
    System.out.println(test.toString());
  }

  public Test() {
    Test test = new Test();
  }
}
```

18.3.3 How many times is the `fib` method in Listing 18.2 invoked for `fib(6)`?

18.4 Problem Solving Using Recursion

Key Point

If you think recursively, you can solve many problems using recursion.

recursion characteristics

The preceding sections presented two classic recursion examples. All recursive methods have the following characteristics:

`if-else`

- The method is implemented using an `if-else` or a `switch` statement that leads to different cases.

base cases

- One or more base cases (the simplest case) are used to stop recursion.

reduction

- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

In general, to solve a problem using recursion, you break it into subproblems. Each subproblem is the same as the original problem, but smaller in size. You can apply the same approach to each subproblem to solve it recursively.

think recursively

Recursion is everywhere. It is fun to *think recursively*. Consider drinking coffee. You may describe the procedure recursively as follows:

```
public static void drinkCoffee(Cup cup) {
  if (!cup.isEmpty()) {
    cup.takeOneSip(); // Take one sip
    drinkCoffee(cup);
  }
}
```

Assume `cup` is an object for a cup of coffee with the instance methods `isEmpty()` and `takeOneSip()`. You can break the problem into two subproblems: one is to drink one sip of coffee, and the other is to drink the rest of the coffee in the cup. The second problem is the same as the original problem, but smaller in size. The base case for the problem is when the cup is empty.

Consider the problem of printing a message `n` times. You can break the problem into two subproblems: one is to print the message one time, and the other is to print it `n - 1` times. The second problem is the same as the original problem, but it is smaller in size. The base case for the problem is `n == 0`. You can solve this problem using recursion as follows:

```
public static void nPrintln(String message, int times) {
  if (times >= 1) {
    System.out.println(message);
    nPrintln(message, times - 1);
  } // The base case is times == 0
}
```

recursive call

Note the `fib` method in the preceding section returns a value to its caller, but the `drinkCoffee` and `nPrintln` methods are `void` and they do not return a value.

think recursively

If you *think recursively*, you can use recursion to solve many of the problems presented in earlier chapters of this book. Consider the palindrome problem in Listing 5.14. Recall that a string is a palindrome if it reads the same from the left and from the right. For example, “mom” and “dad” are palindromes but “uncle” and “aunt” are not. The problem of checking whether a string is a palindrome can be divided into two subproblems:

- Check whether the first character and the last character of the string are equal.
- Ignore the two end characters and check whether the rest of the substring is a palindrome.

The second subproblem is the same as the original problem, but smaller in size. There are two base cases: (1) the two end characters are not the same and (2) the string size is `0` or `1`. In case 1, the string is not a palindrome; in case 2, the string is a palindrome. The recursive method for this problem can be implemented as given in Listing 18.3.

LISTING 18.3 `RecursivePalindromeUsingSubstring.java`

```
public class RecursivePalindromeUsingSubstring {
  public static boolean isPalindrome(String s) {
    if (s.length() <= 1) // Base case
      return true;
    else if (s.charAt(0) != s.charAt(s.length() -  1)) // Base case
      return false;
    else
      return isPalindrome(s.substring(1, s.length() - 1));
  }

  public static void main(String[] args) {
    System.out.println("Is moon a palindrome? "
      + isPalindrome("moon"));
    System.out.println("Is noon a palindrome? "
      + isPalindrome("noon"));
    System.out.println("Is a a palindrome? " + isPalindrome("a"));
    System.out.println("Is aba a palindrome? " +
      isPalindrome("aba"));
    System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
  }
}
```

method header

base case

base case

recursive call

```
Is moon a palindrome? false
Is noon a palindrome? true
Is a a palindrome? true
Is aba a palindrome? true
Is ab a palindrome? false
```

The `substring` method in line 8 creates a new string that is same as the original string except without the first and the last characters. Checking whether a string is a palindrome is equivalent to checking whether the substring is a palindrome if the two end characters in the original string are the same.

Check Point

18.4.1 Describe the characteristics of recursive methods.

18.4.2 For the `isPalindrome` method in Listing 18.3, what are the base cases? How many times is this method called when invoking `isPalindrome("abdxcxdba")`?

18.4.3 Show the call stack for `isPalindrome("abcba")` using the method defined in Listing 18.3.

18.5 Recursive Helper Methods

Key Point

Sometimes you can find a solution to the original problem by defining a recursive function to a problem similar to the original problem. This new method is called a recursive helper method. The original problem can be solved by invoking the recursive helper method.

The recursive `isPalindrome` method in Listing 18.3 is not efficient because it creates a new string for every recursive call. To avoid creating new strings, you can use the low and high indices to indicate the range of the substring. These two indices must be passed to the recursive method. Since the original method is `isPalindrome(String s)`, you have to create the new method `isPalindrome(String s, int low, int high)` to accept additional information on the string, as given in Listing 18.4.

Listing 18.4 RecursivePalindrome.java

helper method
base case

base case

```java
public class RecursivePalindrome {
  public static boolean isPalindrome(String s) {
    return isPalindrome(s, 0, s.length() - 1);
  }

  private static boolean isPalindrome(String s, int low, int high) {
    if (high <= low) // Base case
      return true;
    else if (s.charAt(low) != s.charAt(high)) // Base case
      return false;
    else
      return isPalindrome(s, low + 1, high - 1);
  }

  public static void main(String[] args) {
    System.out.println("Is moon a palindrome? "
      + isPalindrome("moon"));
    System.out.println("Is noon a palindrome? "
      + isPalindrome("noon"));
    System.out.println("Is a a palindrome? " + isPalindrome("a"));
    System.out.println("Is aba a palindrome? " + isPalindrome("aba"));
    System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
  }
}
```

Two overloaded `isPalindrome` methods are defined. The first method `isPalindrome(String s)` checks whether a string is a palindrome and the second method `isPalindrome(String s, int low, int high)` checks whether a substring `s(low..high)` is a palindrome. The first method passes the string `s` with `low = 0` and `high = s.length() – 1` to the second method. The second method can be invoked recursively to check a palindrome in an ever-shrinking substring. It is a common design technique in recursive programming to define a second method that receives additional parameters. Such a method is known as a *recursive helper method*.

recursive helper method

Helper methods are very useful in designing recursive solutions for problems involving strings and arrays. The sections that follow give two more examples.

18.5.1 Recursive Selection Sort

Selection sort was introduced in Section 7.11. Recall that it finds the smallest element in the list and swaps it with the first element. It then finds the smallest element remaining and swaps it with the first element in the remaining list and so on until the remaining list contains only a single element. The problem can be divided into two subproblems:

- Find the smallest element in the list and swap it with the first element.
- Ignore the first element and sort the remaining smaller list recursively.

The base case is that the list contains only one element. Listing 18.5 gives the recursive sort method.

LISTING 18.5 `RecursiveSelectionSort.java`

```java
public class RecursiveSelectionSort {
  public static void sort(double[] list) {
    sort(list, 0, list.length - 1); // Sort the entire list
  }

  private static void sort(double[] list, int low, int high) {
    if (low < high) {
      // Find the smallest number and its index in list[low .. high]
      int indexOfMin = low;
      double min = list[low];
      for (int i = low + 1; i <= high; i++) {
        if (list[i] < min) {
          min = list[i];
          indexOfMin = i;
        }
      }

      // Swap the smallest in list[low .. high] with list[low]
      list[indexOfMin] = list[low];
      list[low] = min;

      // Sort the remaining list[low+1 .. high]
      sort(list, low + 1, high);
    }
  }
}
```

helper method

base case

recursive call

Two overloaded `sort` methods are defined. The first method `sort(double[] list)` sorts an array in `list[0..list.length – 1]` and the second method `sort(double[] list, int low, int high)` sorts an array in `list[low..high]`. The second method can be invoked recursively to sort an ever-shrinking subarray.

VideoNote
Binary search

18.5.2 Recursive Binary Search

Binary search was introduced in Section 7.10.2. For binary search to work, the elements in the array must be in increasing order. The binary search first compares the key with the element in the middle of the array. Consider the following three cases:

- Case 1: If the key is less than the middle element, recursively search for the key in the first half of the array.
- Case 2: If the key is equal to the middle element, the search ends with a match.
- Case 3: If the key is greater than the middle element, recursively search for the key in the second half of the array.

Case 1 and Case 3 reduce the search to a smaller list. Case 2 is a base case when there is a match. Another base case is that the search is exhausted without a match. Listing 18.6 gives a clear, simple solution for the binary search problem using recursion.

LISTING 18.6 `RecursiveBinarySearch.java`

helper method
base case
recursive call
base case
recursive call

```java
public class RecursiveBinarySearch {
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;
    return binarySearch(list, key, low, high);
  }

  private static int binarySearch(int[] list, int key,
      int low, int high) {
    if (low > high) // The list has been exhausted without a match
      return -low - 1;

    int mid = (low + high) / 2;
    if (key < list[mid])
      return binarySearch(list, key, low, mid - 1);
    else if (key == list[mid])
      return mid;
    else
      return binarySearch(list, key, mid + 1, high);
  }
}
```

The first method finds a key in the whole list. The second method finds a key in the list with index from `low` to `high`.

The first `binarySearch` method passes the initial array with `low = 0` and `high = list.length - 1` to the second `binarySearch` method. The second method is invoked recursively to find the key in an ever-shrinking subarray.

Check Point

18.5.1 Show the call stack for `isPalindrome("abcba")` using the method defined in Listing 18.4.

18.5.2 Show the call stack for `selectionSort(new double[]{2, 3, 5, 1})` using the method defined in Listing 18.5.

18.5.3 What is a recursive helper method?

18.6 Case Study: Finding the Directory Size

Recursive methods are efficient for solving problems with recursive structures.

Key Point

The preceding examples can easily be solved without using recursion. This section presents a problem that is difficult to solve without using recursion. The problem is to find the size of a directory. The size of a directory is the sum of the sizes of all files in the directory. A directory d may contain subdirectories. Suppose a directory contains files $f_1, f_2, \ldots, f_m$ and subdirectories $d_1, d_2, \ldots, d_n$, as shown in Figure 18.5.

VideoNote
Directory size

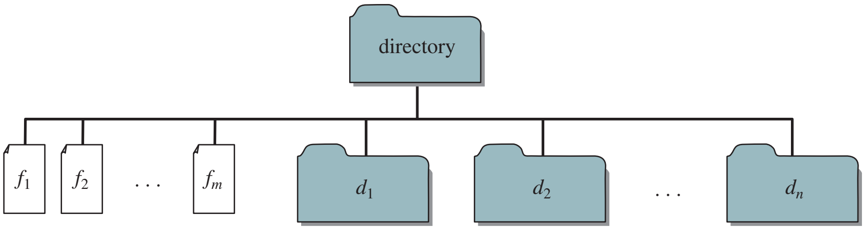


Figure 18.5 A directory contains files and subdirectories.

The size of the directory can be defined recursively as follows:

$$size(d) = size(f_1) + size(f_2) + \cdots + size(f_m) + size(d_1) + size(d_2) + \cdots + size(d_n)$$

The `File` class, introduced in Section 12.10, can be used to represent a file or a directory and obtain the properties for files and directories. Two methods in the `File` class are useful for this problem:

- The `length()` method returns the size of a file.
- The `listFiles()` method returns an array of `File` objects under a directory.

Listing 18.7 gives a program that prompts the user to enter a directory or a file and displays its size.

Listing 18.7 DirectorySize.java

```
import java.io.File;
import java.util.Scanner;

public class DirectorySize {
  public static void main(String[] args) {
    // Prompt the user to enter a directory or a file
    System.out.print("Enter a directory or a file: ");
    Scanner input = new Scanner(System.in);
    String directory = input.nextLine();

    // Display the size
    System.out.println(getSize(new File(directory)) + " bytes");
  }

  public static long getSize(File file) {
    long size = 0; // Store the total size of all files

```

invoke method

getSize method

is directory?
all subitems
recursive call
base case

```
18    if (file.isDirectory()) {
19      File[] files = file.listFiles(); // All files and subdirectories
20      for (int i = 0; files != null && i < files.length; i++) {
21        size += getSize(files[i]); // Recursive call
22      }
23    }
24    else { // Base case
25      size += file.length();
26    }
27
28    return size;
29  }
30 }
```

```
Enter a directory or a file: c:\book ↵Enter
48619631 bytes
```

```
Enter a directory or a file: c:\book\Welcome.java ↵Enter
172 bytes
```

```
Enter a directory or a file: c:\book\NonExistentFile ↵Enter
0 bytes
```

If the **`file`** object represents a directory (line 18), each subitem (file or subdirectory) in the directory is recursively invoked to obtain its size (line 21). If the **`file`** object represents a file (line 24), the file size is obtained and added to the total size (line 25).

What happens if an incorrect or a nonexistent directory is entered? The program will detect that it is not a directory and invoke **`file.length()`** (line 25), which returns **`0`**. Thus, in this case, the **`getSize`** method will return **`0`**.

testing all cases

Tip

To avoid mistakes, it is a good practice to test all cases. For example, you should test the program for an input of file, an empty directory, a nonexistent directory, and a nonexistent file.

Check Point

18.6.1 What is the base case for the **`getSize`** method?

18.6.2 How does the program get all files and directories under a given directory?

18.6.3 How many times will the **`getSize`** method be invoked for a directory if the directory has three subdirectories and each subdirectory has four files?

18.6.4 Will the program work if the directory is empty (i.e., it does not contain any files)?

18.6.5 Will the program work if line 20 is replaced by the following code?

```
for (int i = 0; i < files.length; i++)
```

18.6.6 Will the program work if lines 20 and 21 are replaced by the following code?

```
for (File file: files)
  size += getSize(file); // Recursive call
```

18.7 Case Study: Tower of Hanoi

The Tower of Hanoi problem is a classic problem that can be solved easily using recursion, but it is difficult to solve otherwise.

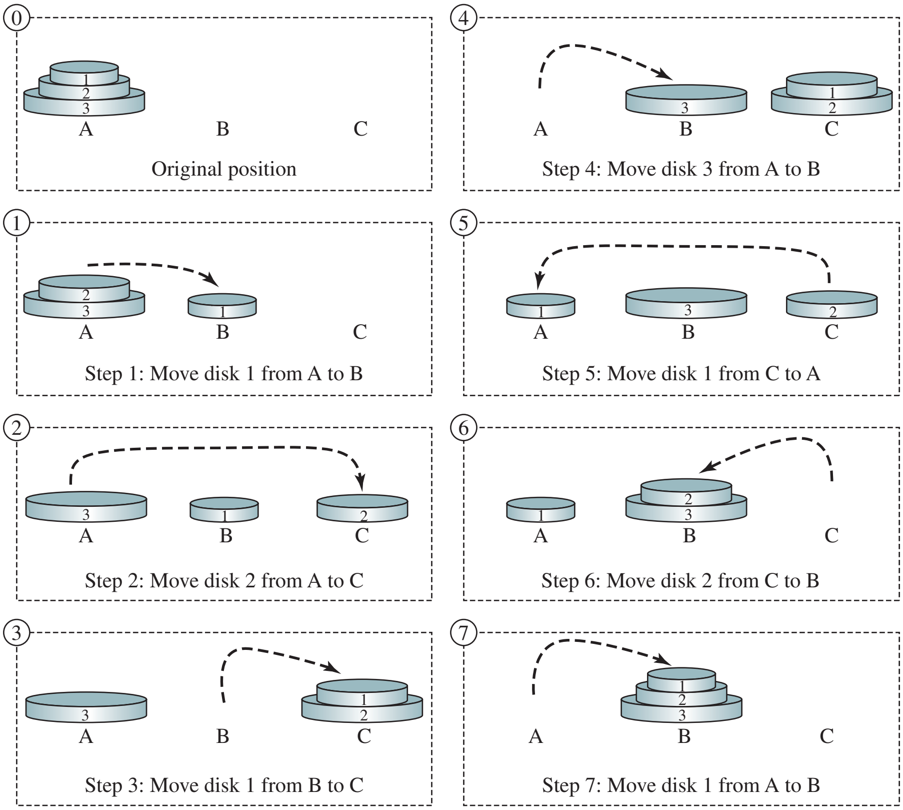
Key Point

The problem involves moving a specified number of disks of distinct sizes from one tower to another while observing the following rules:

- There are *n* disks labeled 1, 2, 3, . . . , *n* and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time and it must be the smallest disk on a tower.

The objective of the problem is to move all the disks from A to B with the assistance of C. For example, if you have three disks, the steps to move all of the disks from A to B are shown in Figure 18.6. For an interactive demo, see liveexample.pearsoncmg.com/dsanimation/TowerOfHanoieBook.html

Tower of Hanoi on Companion Website



(0)
A
B
C
Original position
(1)
Step 1: Move disk 1 from A to B
(2)
Step 2: Move disk 2 from A to C
(3)
Step 3: Move disk 1 from B to C
(4)
Step 4: Move disk 3 from A to B
(5)
Step 5: Move disk 1 from C to A
(6)
Step 6: Move disk 2 from C to B
(7)
Step 7: Move disk 1 from A to B

FIGURE 18.6 The goal of the Tower of Hanoi problem is to move disks from tower A to tower B without breaking the rules.

In the case of three disks, you can find the solution manually. For a larger number of disks, however—even for four—the problem is quite complex. Fortunately, the problem has an inherently recursive nature, which leads to a straightforward recursive solution.

The base case for the problem is `n = 1`. If `n == 1`, you could simply move the disk from A to B. When `n > 1`, you could split the original problem into the following three subproblems and solve them sequentially.

1. Move the first `n - 1` disks from A to C recursively with the assistance of tower B, as shown in Step 1 in Figure 18.7.
2. Move disk `n` from A to B, as shown in Step 2 in Figure 18.7.
3. Move `n - 1` disks from C to B recursively with the assistance of tower A, as shown in Step 3 in Figure 18.7.

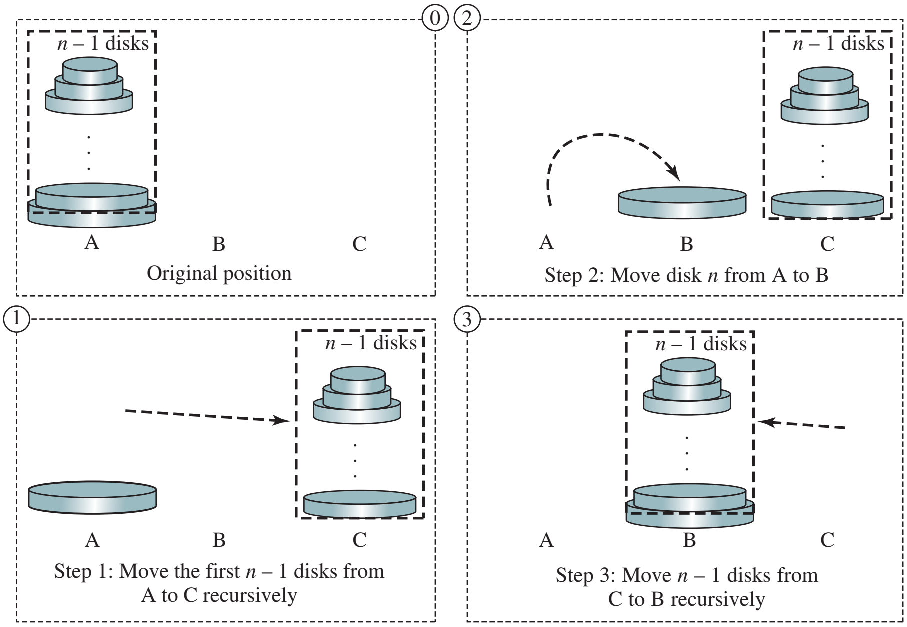

⓪
n – 1 disks
A
B
C
Original position
②
n – 1 disks
A
B
C
Step 2: Move disk *n* from A to B
①
n – 1 disks
A
B
C
Step 1: Move the first *n* – 1 disks from A to C recursively
③
n – 1 disks
A
B
C
Step 3: Move *n* – 1 disks from C to B recursively

FIGURE 18.7 The Tower of Hanoi problem can be decomposed into three subproblems.

The following method moves *n* disks from the `fromTower` to the `toTower` with the assistance of the `auxTower`:

```
void moveDisks(int n, char fromTower, char toTower, char auxTower)
```

The algorithm for the method can be described as

```
if (n == 1) // Stopping condition
  Move disk 1 from the fromTower to the toTower;
else {
  moveDisks(n - 1, fromTower, auxTower, toTower);
  Move disk n from the fromTower to the toTower;
  moveDisks(n - 1, auxTower, toTower, fromTower);
}
```

Listing 18.8 gives a program that prompts the user to enter the number of disks and invokes the recursive method `moveDisks` to display the solution for moving the disks.

LISTING 18.8 TowerOfHanoi.java

```
import java.util.Scanner;

public class TowerOfHanoi {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter number of disks: ");
    int n = input.nextInt();

    // Find the solution recursively
    System.out.println("The moves are:");
    moveDisks(n, 'A', 'B', 'C');
  }

  /** The method for finding the solution to move n disks
      from fromTower to toTower with auxTower */
  public static void moveDisks(int n, char fromTower,
      char toTower, char auxTower) {
    if (n == 1) // Stopping condition
      System.out.println("Move disk " + n + " from " +
        fromTower + " to " + toTower);
    else {
      moveDisks(n - 1, fromTower, auxTower, toTower);
      System.out.println("Move disk " + n + " from " +
        fromTower + " to " + toTower);
      moveDisks(n - 1, auxTower, toTower, fromTower);
    }
  }
}
```

base case

recursion

recursion

```
Enter number of disks: 4
The moves are:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Move disk 4 from A to B
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A
Move disk 3 from C to B
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
```

This problem is inherently recursive. Using recursion makes it possible to find a natural, simple solution. It would be difficult to solve the problem without using recursion.

Consider tracing the program for `n = 3`. The successive recursive calls are shown in Figure 18.8. As you can see, writing the program is easier than tracing the recursive calls. The system uses stacks to manage the calls behind the scenes. To some extent, recursion provides a level of abstraction that hides iterations and other details from the user.

Check Point

18.7.1 How many times is the `moveDisks` method in Listing 18.8 invoked for `moveDisks(5, 'A', 'B', 'C')`?

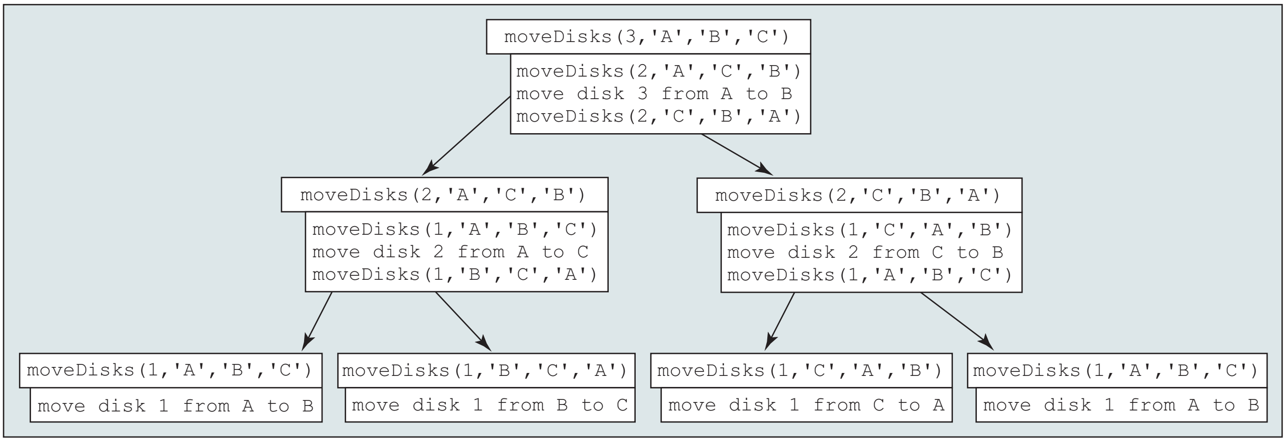

moveDisks(3,'A','B','C')
moveDisks(2,'A','C','B')
move disk 3 from A to B
moveDisks(2,'C','B','A')
moveDisks(2,'A','C','B')
moveDisks(1,'A','B','C')
move disk 2 from A to C
moveDisks(1,'B','C','A')
moveDisks(2,'C','B','A')
moveDisks(1,'C','A','B')
move disk 2 from C to B
moveDisks(1,'A','B','C')
moveDisks(1,'A','B','C')
move disk 1 from A to B
moveDisks(1,'B','C','A')
move disk 1 from B to C
moveDisks(1,'C','A','B')
move disk 1 from C to A
moveDisks(1,'A','B','C')
move disk 1 from A to B

FIGURE 18.8 Invoking `moveDisks(3, 'A', 'B', 'C')` spawns calls to `moveDisks` recursively.

18.8 Case Study: Fractals

Key Point

Using recursion is ideal for displaying fractals because fractals are inherently recursive.

A *fractal* is a geometrical figure, but unlike triangles, circles, and rectangles, fractals can be divided into parts, each of which is a reduced-size copy of the whole. There are many interesting examples of fractals. This section introduces a simple fractal, the *Sierpinski triangle*, named after a famous Polish mathematician.

A Sierpinski triangle is created as follows:

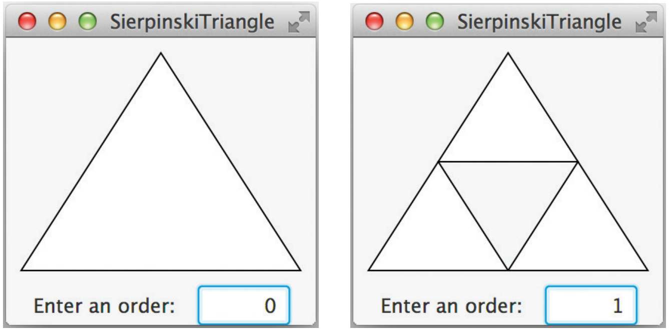
1. Begin with an equilateral triangle, which is considered to be a Sierpinski fractal of order (or level) `0`, as shown in Figure 18.9a.
2. Connect the midpoints of the sides of the triangle of order `0` to create a Sierpinski triangle of order `1` (see Figure 18.9b).
3. Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski triangle of order `2` (see Figure 18.9c).
4. You can repeat the same process recursively to create a Sierpinski triangle of order `3`, `4`, . . . , and so on (see Figure 18.9d). For an interactive demo, see liveexample.pearsoncmg.com/dsanimation/SierpinskiTriangleUsingHTML.html.

The problem is inherently recursive. How do you develop a recursive solution for it? Consider the base case when the order is `0`. It is easy to draw a Sierpinski triangle of order `0`. How do you draw a Sierpinski triangle of order `1`? The problem can be reduced to drawing three Sierpinski triangles of order `0`. How do you draw a Sierpinski triangle of order `2`? The problem can be reduced to drawing three Sierpinski triangles of order `1`, so the problem of drawing a Sierpinski triangle of order `n` can be reduced to drawing three Sierpinski triangles of order `n - 1`.

Listing 18.9 gives a program that displays a Sierpinski triangle of any order, as shown in Figure 18.9. You can enter an order in a text field to display a Sierpinski triangle of the specified order.

LISTING 18.9 `SierpinskiTriangle.java`

```java
import javafx.application.Application;
import javafx.geometry.Point2D;
import javafx.geometry.Pos;
```

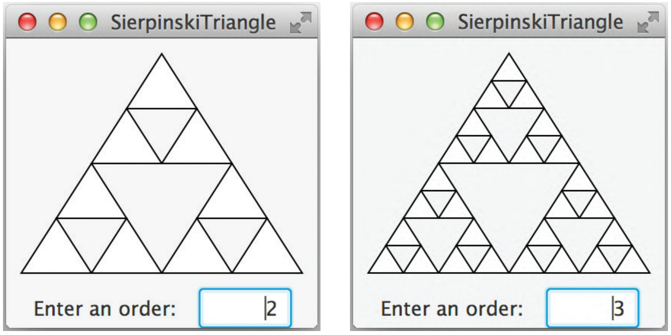


Enter an order: 0

Enter an order: 1

(a) Order 0

(b) Order 1



Enter an order: 2

Enter an order: 3

(c) Order 2

(d) Order 3

FIGURE 18.9 A Sierpinski triangle is a pattern of recursive triangles. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

```
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Polygon;
import javafx.stage.Stage;

public class SierpinskiTriangle extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    SierpinskiTrianglePane pane = new SierpinskiTrianglePane();
    TextField tfOrder = new TextField();
    tfOrder.setOnAction(
      e -> pane.setOrder(Integer.parseInt(tfOrder.getText())));
    tfOrder.setPrefColumnCount(4);
    tfOrder.setAlignment(Pos.BOTTOM_RIGHT);

    // Pane to hold label, text field, and a button
    HBox hBox = new HBox(10);
    hBox.getChildren().addAll(new Label("Enter an order: "), tfOrder);
    hBox.setAlignment(Pos.CENTER);

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(pane);
    borderPane.setBottom(hBox);

```

recursive triangle pane

listener for text field

hold label and text field

listener for resizing

three initial points

clear the pane

draw a triangle

create a triangle

top subtriangle

left subtriangle

right subtriangle

```
  // Create a scene and place it in the stage
  Scene scene = new Scene(borderPane, 200, 210);
  primaryStage.setTitle("SierpinskiTriangle"); // Set the stage title
  primaryStage.setScene(scene); // Place the scene in the stage
  primaryStage.show(); // Display the stage

  pane.widthProperty().addListener(ov -> pane.paint());
  pane.heightProperty().addListener(ov -> pane.paint());
 }

 /** Pane for displaying triangles */
 static class SierpinskiTrianglePane extends Pane {
  private int order = 0;

  /** Set a new order */
  public void setOrder(int order) {
   this.order = order;
   paint();
  }

  SierpinskiTrianglePane() {
  }

  protected void paint() {
   // Select three points in proportion to the pane size
   Point2D p1 = new Point2D(getWidth() / 2, 10);
   Point2D p2 = new Point2D(10, getHeight() - 10);
   Point2D p3 = new Point2D(getWidth() - 10, getHeight() - 10);

   this.getChildren().clear(); // Clear the pane before redisplay

   displayTriangles(order, p1, p2, p3);
  }

  private void displayTriangles(int order, Point2D p1,
    Point2D p2, Point2D p3) {
   if (order == 0) {
    // Draw a triangle to connect three points
    Polygon triangle = new Polygon();
    triangle.getPoints().addAll(p1.getX(), p1.getY(), p2.getX(),
      p2.getY(), p3.getX(), p3.getY());
    triangle.setStroke(Color.BLACK);
    triangle.setFill(Color.WHITE);

    this.getChildren().add(triangle);
   }
   else {
    // Get the midpoint on each edge in the triangle
    Point2D p12 = p1.midpoint(p2);
    Point2D p23 = p2.midpoint(p3);
    Point2D p31 = p3.midpoint(p1);

    // Recursively display three triangles
    displayTriangles(order - 1, p1, p12, p31);
    displayTriangles(order - 1, p12, p2, p23);
    displayTriangles(order - 1, p31, p23, p3);
   }
  }
 }
}
```

The initial triangle has three points set in proportion to the pane size (lines 58–60). If `order == 0`, the `displayTriangles(order, p1, p2, p3)` method displays a triangle that connects the three points `p1`, `p2`, and `p3` (lines 71–77), as shown in Figure 18.10a. Otherwise, it performs the following tasks:

`displayTriangle` method

1. Obtain the midpoint between `p1` and `p2` (line 81), the midpoint between `p2` and `p3` (line 82), and the midpoint between `p3` and `p1` (line 83), as shown in Figure 18.10b.
2. Recursively invoke `displayTriangles` with a reduced order to display three smaller Sierpinski triangles (lines 86–88). Note each small Sierpinski triangle is structurally identical to the original big Sierpinski triangle except that the order of a small triangle is one less, as shown in Figure 18.10b.

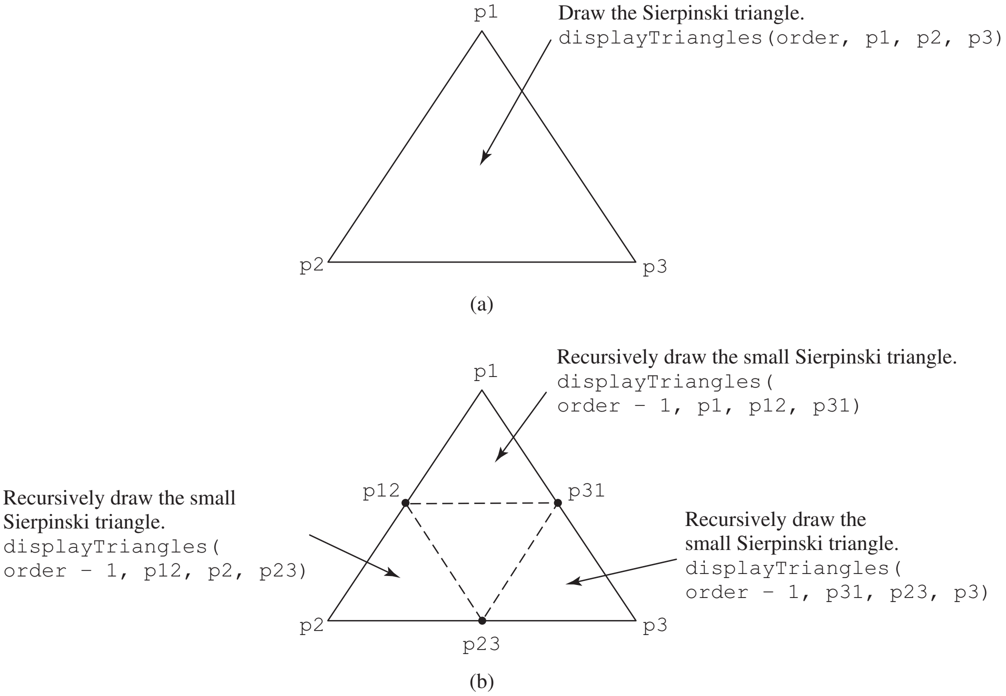

p1
Draw the Sierpinski triangle.
displayTriangles(order, p1, p2, p3)
p2
p3
(a)
Recursively draw the small Sierpinski triangle.
displayTriangles(
order - 1, p1, p12, p31)
p1
p12
p31
Recursively draw the small
Sierpinski triangle.
displayTriangles(
order - 1, p12, p2, p23)
Recursively draw the
small Sierpinski triangle.
displayTriangles(
order - 1, p31, p23, p3)
p2
p23
p3
(b)

FIGURE 18.10 Drawing a Sierpinski triangle spawns calls to draw three small Sierpinski triangles recursively.

A Sierpinski triangle is displayed in a `SierpinskiTrianglePane`. The `order` property in the inner class `SierpinskiTrianglePane` specifies the order for the Sierpinski triangle. The `Point2D` class, introduced in Section 9.6.3, The `Point2D` Class, represents a point with *x*- and *y*-coordinates. Invoking `p1.midpoint(p2)` returns a new `Point2D` object that is the midpoint between `p1` and `p2` (lines 81–83).

Check Point

18.8.1 How do you obtain the midpoint between two points?

18.8.2 What is the base case for the `displayTriangles` method?

18.8.3 How many times is the `displayTriangles` method invoked for a Sierpinski triangle of order 0, order 1, order 2, and order *n*?

18.8.4 What happens if you enter a negative order? How do you fix this problem in the code?

18.8.5 Instead of drawing a triangle using a polygon, rewrite the code to draw a triangle by drawing three lines to connect the points in lines 71–77.

18.9 Recursion vs. Iteration

Key Point

Recursion is an alternative form of program control. It is essentially repetition without a loop.

When you use loops, you specify a loop body. The repetition of the loop body is controlled by the loop control structure. In recursion, the method itself is called repeatedly. A selection statement must be used to control whether to call the method recursively or not.

recursion overhead

Recursion bears substantial overhead. Each time the program calls a method, the system must allocate memory for all of the method's local variables and parameters. This can consume considerable memory and requires extra time to manage the memory.

recursion advantages

Any problem that can be solved recursively can be solved nonrecursively with iterations. Recursion has some negative aspects: it uses up too much time and too much memory. Why, then, should you use it? In some cases, using recursion enables you to specify a clear, simple solution for an inherently recursive problem that would otherwise be difficult to obtain. Examples are the directory-size problem, the Tower of Hanoi problem, and the fractal problem, which are rather difficult to solve without using recursion.

recursion or iteration?

The decision whether to use recursion or iteration should be based on the nature of, and your understanding of, the problem you are trying to solve. The rule of thumb is to use whichever approach can best develop an intuitive solution that naturally mirrors the problem. If an iterative solution is obvious, use it. It will generally be more efficient than the recursive option.

`StackOverflowError`

Note

Recursive programs can run out of memory, causing a `StackOverflowError`.

performance concern

Tip

If you are concerned about your program's performance, avoid using recursion because it takes more time and consumes more memory than iteration. In general, recursion can be used to solve the inherent recursive problems such as Tower of Hanoi, recursive directories, and Sierpinski triangles.

Check Point

18.9.1 Which of the following statements are true?

a. Any recursive method can be converted into a nonrecursive method.

b. Recursive methods take more time and memory to execute than nonrecursive methods.

c. Recursive methods are *always* simpler than nonrecursive methods.

d. There is always a selection statement in a recursive method to check whether a base case is reached.

18.9.2 What is a cause for a stack-overflow exception?

18.10 Tail Recursion

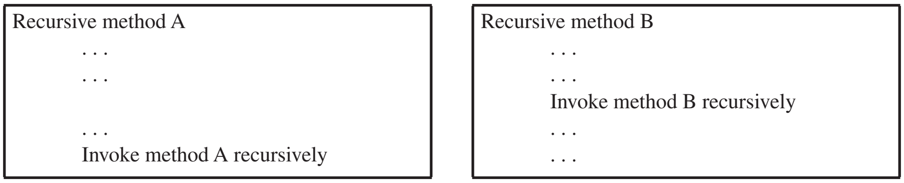
Key Point

A tail-recursive method is efficient.

tail recursion

A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call, as illustrated in Figure 18.11a. However, method `B` in Figure 18.11b is not tail recursive because there are pending operations after a method call is returned.

For example, the recursive `isPalindrome` method (lines 6–13) in Listing 18.4 is tail recursive because there are no pending operations after recursively invoking `isPalindrome` in line 12. However, the recursive `factorial` method (lines 16–21) in Listing 18.1 is not tail recursive because there is a pending operation, namely multiplication, to be performed on return from each recursive call.



(a) Tail recursion (b) Nontail recursion

FIGURE 18.11 A tail-recursive method has no pending operations after a recursive call.

Tail recursion is desirable because the method ends when the last recursive call ends, and there is no need to store the intermediate calls in the stack. Compilers can optimize tail recursion to reduce stack size.

A nontail-recursive method can often be converted to a tail-recursive method by using auxiliary parameters. These parameters are used to contain the result. The idea is to incorporate the pending operations into the auxiliary parameters in such a way that the recursive call no longer has a pending operation. You can define a new auxiliary recursive method with the auxiliary parameters. This method may overload the original method with the same name but a different signature. For example, the `factorial` method in Listing 18.1 is written in a tail-recursive way in Listing 18.10.

LISTING 18.10 ComputeFactorialTailRecursion.java

```
 1  public class ComputeFactorialTailRecursion {
 2    /** Return the factorial for a specified number */
 3    public static long factorial(int n) {
 4      return factorial(n, 1); // Call auxiliary method
 5    }
 6
 7    /** Auxiliary tail-recursive method for factorial */
 8    private static long factorial(int n, int result) {
 9      if (n == 0)
10        return result;
11      else
12        return factorial(n - 1, n * result); // Recursive call
13    }
14  }
```

original method
invoke auxiliary method
auxiliary method
recursive call

The first `factorial` method (line 3) simply invokes the second auxiliary method (line 4). The second method contains an auxiliary parameter `result` that stores the result for the factorial of `n`. This method is invoked recursively in line 12. There is no pending operation after a call is returned. The final result is returned in line 10, which is also the return value from invoking `factorial(n, 1)` in line 4.

18.10.1 Identify tail-recursive methods in this chapter.

18.10.2 Rewrite the `fib` method in Listing 18.2 using tail recursion.

Check
Point

KEY TERMS

base case	720
direct recursion	723
indirect recursion	723
infinite recursion	723
recursive helper method	729
recursive method	720
stopping condition	720
tail recursion	740

CHAPTER SUMMARY

1. A *recursive method* is one that directly or indirectly invokes itself. For a recursive method to terminate, there must be one or more *base cases*.
2. *Recursion* is an alternative form of program control. It is essentially repetition without a loop control. It can be used to write simple, clear solutions for inherently recursive problems that would otherwise be difficult to solve.
3. Sometimes the original method needs to be modified to receive additional parameters in order to be invoked recursively. A *recursive helper method* can be defined for this purpose.
4. Recursion bears substantial overhead. Each time the program calls a method, the system must allocate memory for all of the method's local variables and parameters. This can consume considerable memory and requires extra time to manage the memory.
5. A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call. Some compilers can optimize tail recursion to reduce stack size.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

Sections 18.2 and 18.3

***18.1** (*Factorial*) Using the `BigInteger` class introduced in Section 10.9, you can find the factorial for a large number (e.g., `100!`). Implement the `factorial` method using recursion. Write a program that prompts the user to enter an integer and displays its factorial.

***18.2** (*Fibonacci numbers*) Rewrite the `fib` method in Listing 18.2 using iterations.

Hint: To compute `fib(n)` without recursion, you need to obtain `fib(n - 2)` and `fib(n - 1)` first. Let `f0` and `f1` denote the two previous Fibonacci numbers. The current Fibonacci number would then be `f0 + f1`. The algorithm can be described as follows:

```
f0 = 0; // For fib(0)
f1 = 1; // For fib(1)

for (int i = 1; i <= n; i++) {
  currentFib = f0 + f1;
  f0 = f1;
  f1 = currentFib;
}
// After the loop, currentFib is fib(n)
```

Write a test program that prompts the user to enter an index and displays its Fibonacci number.

***18.3** (*Compute greatest common divisor using recursion*) The `gcd(m, n)` can also be defined recursively as follows:

- If `m % n` is `0`, `gcd(m, n)` is `n`.
- Otherwise, `gcd(m, n)` is `gcd(n, m % n)`.

Write a recursive method to find the GCD. Write a test program that prompts the user to enter two integers and displays their GCD.

18.4 (*Sum series*) Write a recursive method to compute the following series:

$$m(i) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i}$$

Write a test program that displays `m(i)` for `i` = `1`, `2`, . . . , `10`.

18.5 (*Sum series*) Write a recursive method to compute the following series:

$$m(i) = \frac{1}{3} + \frac{2}{5} + \frac{3}{7} + \frac{4}{9} + \frac{5}{11} + \frac{6}{13} + \cdots + \frac{i}{2i + 1}$$

Write a test program that displays `m(i)` for `i = 1`, `2`, . . . , `10`.

***18.6** (*Sum series*) Write a recursive method to compute the following series:

$$m(i) = \frac{1}{2} + \frac{2}{3} + \cdots + \frac{i}{i + 1}$$

Write a test program that displays `m(i)` for `i` = `1`, `2`, . . . , `10`.

***18.7** (*Fibonacci series*) Modify Listing 18.2, ComputeFibonacci.java, so that the program finds the number of times the `fib` method is called. (*Hint*: Use a static variable and increment it every time the method is called.)

Section 18.4

***18.8** (*Print the digits in an integer reversely*) Write a recursive method that displays an `int` value reversely on the console using the following header:

```
public static void reverseDisplay(int value)
```

For example, `reverseDisplay(12345)` displays `54321`. Write a test program that prompts the user to enter an integer and displays its reversal.

***18.9** (*Print the characters in a string reversely*) Write a recursive method that displays a string reversely on the console using the following header:

```
public static void reverseDisplay(String value)
```

For example, `reverseDisplay("abcd")` displays `dcba`. Write a test program that prompts the user to enter a string and displays its reversal.

***18.10** (*Occurrences of a specified character in a string*) Write a recursive method that finds the number of occurrences of a specified letter in a string using the following method header:

```
public static int count(String str, char a)
```

For example, `count("Welcome", 'e')` returns `2`. Write a test program that prompts the user to enter a string and a character, and displays the number of occurrences for the character in the string.

***18.11** (*Sum the digits in an integer using recursion*) Write a recursive method that computes the sum of the digits in an integer. Use the following method header:

```
public static int sumDigits(long n)
```

For example, `sumDigits(234)` returns 2 + 3 + 4 = 9. Write a test program that prompts the user to enter an integer and displays its sum.

Section 18.5

****18.12** (*Print the characters in a string reversely*) Rewrite Programming Exercise 18.9 using a helper method to pass the substring `high` index to the method. The helper method header is

```
public static void reverseDisplay(String value, int high)
```

***18.13** (*Find the largest number in an array*) Write a recursive method that returns the largest integer in an array. Write a test program that prompts the user to enter a list of eight integers and displays the largest element.

***18.14** (*Find the number of uppercase letters in a string*) Write a recursive method to return the number of uppercase letters in a string. Write a test program that prompts the user to enter a string and displays the number of uppercase letters in the string.

***18.15** (*Occurrences of a specified character in a string*) Rewrite Programming Exercise 18.10 using a helper method to pass the substring `high` index to the method. The helper method header is

```
public static int count(String str, char a, int high)
```

***18.16** (*Find the number of uppercase letters in an array*) Write a recursive method to return the number of uppercase letters in an array of characters. You need to define the following two methods. The second one is a recursive helper method.

```
public static int count(char[] chars)
public static int count(char[] chars, int high)
```

Write a test program that prompts the user to enter a list of characters in one line and displays the number of uppercase letters in the list.

***18.17** (*Occurrences of a specified character in an array*) Write a recursive method that finds the number of occurrences of a specified character in an array. You need to define the following two methods. The second one is a recursive helper method.

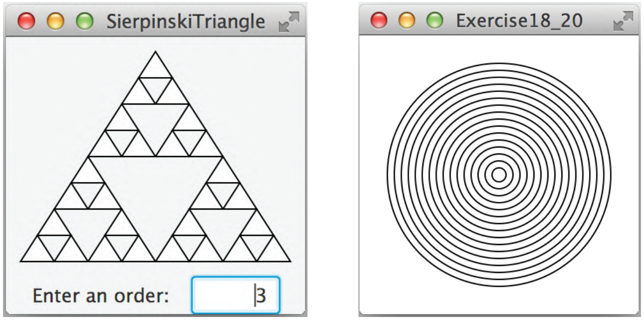
```
public static int count(char[] chars, char ch)
public static int count(char[] chars, char ch, int high)
```

Write a test program that prompts the user to enter a list of characters in one line, and a character, and displays the number of occurrences of the character in the list.

Sections 18.6–18.10

***18.18** (*Tower of Hanoi*) Modify Listing 18.8, TowerOfHanoi.java, so the program finds the number of moves needed to move n disks from tower A to tower B. (*Hint*: Use a static variable and increment it every time the method is called.)

***18.19** (*Sierpinski triangle*) Revise Listing 18.9 to develop a program that lets the user use the +/− buttons, primary/secondary mouse buttons, and UP/DOWN arrow keys to increase or decrease the current order by `1`, as shown in Figure 18.12a. The initial order is **0**. If the current order is **0**, the *Decrease* button is ignored.



(a)

(b)

FIGURE 18.12 (a) Programming Exercise 18.19 uses the + or − buttons to increase or decrease the current order by **1**. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) Programming Exercise 18.20 draws ovals using a recursive method.

***18.20** (*Display circles*) Write a Java program that displays ovals, as shown in Figure 18.12b. The circles are centered in the pane. The gap between two adjacent circles is **10** pixels, and the gap between the border of the pane and the largest circle is also **10**.

***18.21** (*Decimal to binary*) Write a recursive method that converts a decimal number into a binary number as a string. The method header is

```
public static String dec2Bin(int value)
```

Write a test program that prompts the user to enter a decimal number and displays its binary equivalent.

***18.22** (*Decimal to hex*) Write a recursive method that converts a decimal number into a hex number as a string. The method header is

```
public static String dec2Hex(int value)
```

Write a test program that prompts the user to enter a decimal number and displays its hex equivalent.

***18.23** (*Binary to decimal*) Write a recursive method that parses a binary number as a string into a decimal integer. The method header is

```
public static int bin2Dec(String binaryString)
```

Write a test program that prompts the user to enter a binary string and displays its decimal equivalent.

***18.24** (*Hex to decimal*) Write a recursive method that parses a hex number as a string into a decimal integer. The method header is

```
public static int hex2Dec(String hexString)
```

Write a test program that prompts the user to enter a hex string and displays its decimal equivalent.

****18.25** (*String permutation*) Write a recursive method to print all the permutations of a string. For example, for the string `abc`, the permutation is

```
abc
acb
bac
bca
cab
cba
```

(*Hint*: Define the following two methods. The second is a helper method.)

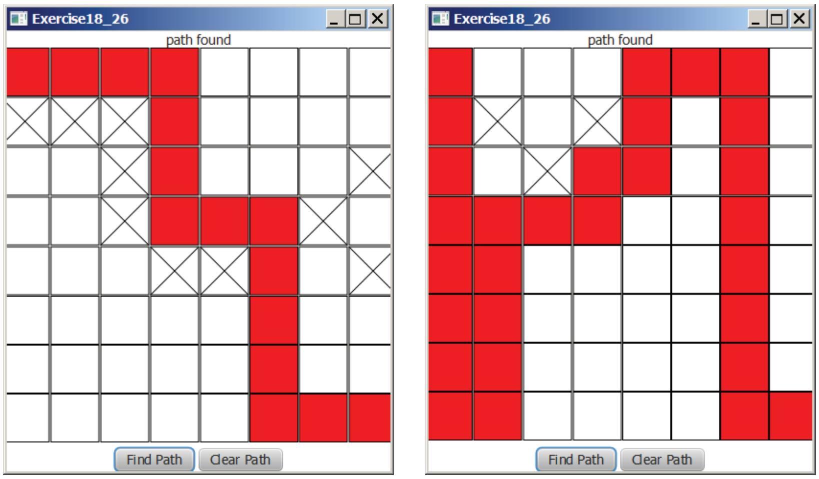
```
public static void displayPermutation(String s)
public static void displayPermutation(String s1, String s2)
```

The first method simply invokes `displayPermutation(" ", s)`. The second method uses a loop to move a character from `s2` to `s1` and recursively invokes it with new `s1` and `s2`. The base case is that `s2` is empty and prints `s1` to the console.

Write a test program that prompts the user to enter a string and displays all its permutations.

****18.26** (*Create a maze*) Write a program that will find a path in a maze, as shown in Figure 18.13a. The maze is represented by a 8 × 8 board. The path must meet the following conditions:

- The path is between the upper-left corner cell and the lower-right corner cell in the maze.



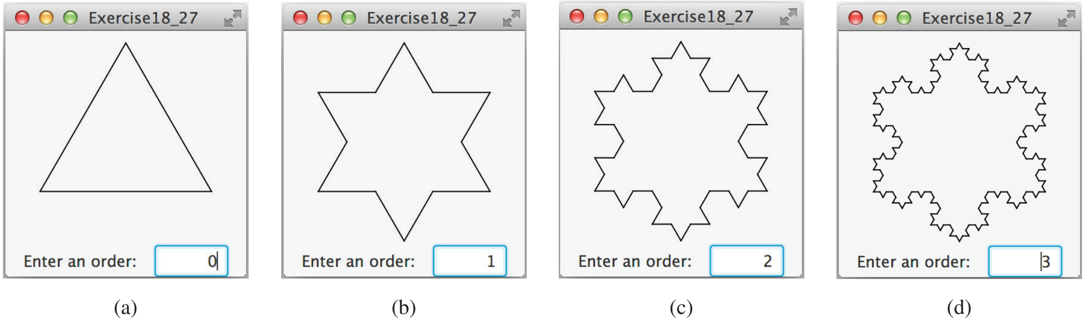
(a) Correct path (b) Illegal path

FIGURE 18.13 The program finds a path from the upper-left corner to the bottom-right corner. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

- The program enables the user to place or remove a mark on a cell. A path consists of adjacent unmarked cells. Two cells are said to be adjacent if they are horizontal or vertical neighbors.
- The path does not contain cells that form a square. The path in Figure 18.13b, for example, does not meet this condition. (The condition makes a path easy to identify on the board.)

****18.27** (*Koch snowflake fractal*) The text presented the Sierpinski triangle fractal. In this exercise, you will write a program to display another fractal, called the *Koch snowflake*, named after a famous Swedish mathematician. A Koch snowflake is created as follows:

1. Begin with an equilateral triangle, which is considered to be the Koch fractal of order (or level) **0**, as shown in Figure 18.14a.
2. Divide each line in the shape into three equal line segments and draw an outward equilateral triangle with the middle line segment as the base to create a Koch fractal of order **1**, as shown in Figure 18.14b.
3. Repeat Step 2 to create a Koch fractal of order **2**, **3**, . . . , and so on, as shown in Figures 18.14c and d.



Enter an order: 0

Enter an order: 1

Enter an order: 2

Enter an order: 3

(a) (b) (c) (d)

Figure 18.14 A Koch snowflake is a fractal starting with a triangle. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****18.28** (*Nonrecursive directory size*) Rewrite Listing 18.7, DirectorySize.java, without using recursion.

***18.29** (*Number of files in a directory*) Write a program that prompts the user to enter a directory and displays the number of the files in the directory.

****18.30** (*Find words*) Write a program that finds all occurrences of a word in all the files under a directory, recursively. Pass the parameters from the command line as follows:

```
java Exercise18_30 dirName word
```

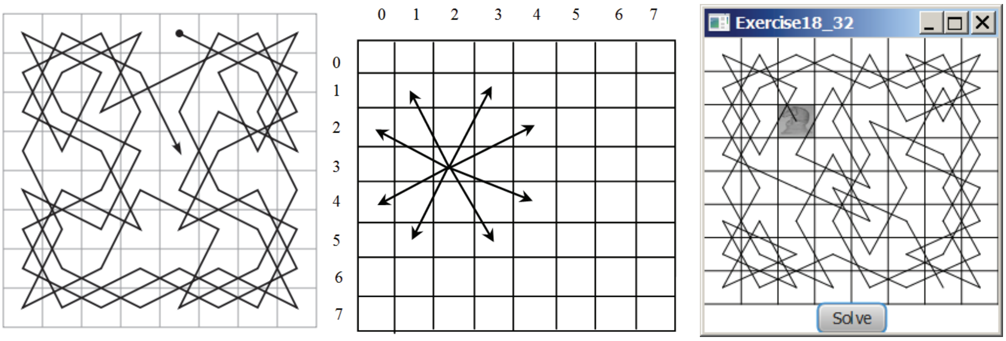
VideoNote
Search a string in a directory

****18.31** (*Replace words*) Write a program that replaces all occurrences of a word with a new word in all the files under a directory, recursively. Pass the parameters from the command line as follows:

```
java Exercise18_31 dirName oldWord newWord
```

*****18.32** (*Game: Knight's Tour*) The Knight's Tour is an ancient puzzle. The objective is to move a knight, starting from any square on a chessboard, to every other square once, as shown in Figure 18.15a. Note the knight makes only L-shaped moves (two spaces in one direction and one space in a perpendicular direction). As shown in Figure 18.15b, the knight can move to eight squares. Write a program that displays the moves for the knight, as shown in Figure 18.15c. When you click a cell, the knight is placed at the cell. This cell will be the starting point for the knight. Click the *Solve* button to display the path for a solution.

(*Hint*: A brute-force approach for this problem is to move the knight from one square to another available square arbitrarily. Using such an approach, your program will take a long time to finish. A better approach is to employ some



(a) (b) (c)

FIGURE 18.15 (a) A knight traverses all squares once. (b) A knight makes an L-shaped move. (c) A program displays a Knight's Tour path. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

heuristics. A knight has two, three, four, six, or eight possible moves, depending on its location. Intuitively, you should attempt to move the knight to the least accessible squares first and leave those more accessible squares open, so there will be a better chance of success at the end of the search.)

*****18.33** (*Game: Knight's Tour animation*) Write a program for the Knight's Tour problem. Your program should let the user move a knight to any starting square and click the *Solve* button to animate a knight moving along the path, as shown in Figure 18.16.

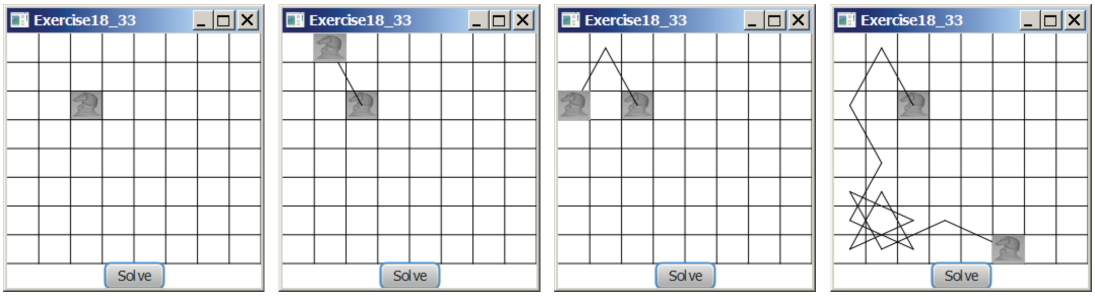


FIGURE 18.16 A knight traverses along the path. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****18.34** (*Game: Eight Queens*) The Eight Queens problem is to find a solution to place a queen in each row on a chessboard such that no two queens can attack each other. Write a program to solve the Eight Queens problem using recursion and display the result as shown in Figure 18.17.

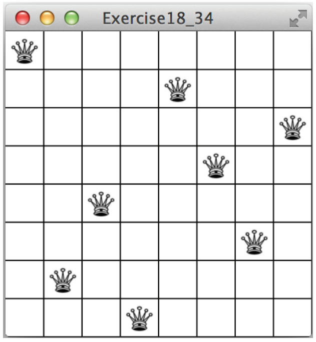


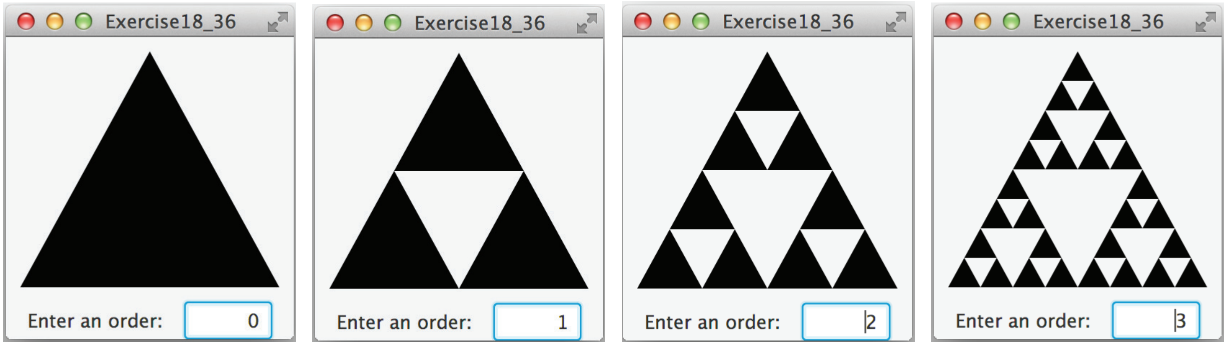
FIGURE 18.17 The program displays a solution to the Eight Queens problem. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****18.35** (*H-tree fractal*) An H-tree (introduced at the beginning of this chapter in Figure 18.1) is a fractal defined as follows:

1. Begin with a letter H. The three lines of H are of the same length, as shown in Figure 18.1a.
2. The letter H (in its sans-serif form, H) has four endpoints. Draw an H centered at each of the four endpoints to an H-tree of order **1**, as shown in Figure 18.1b. These Hs are half the size of the H that contains the four endpoints.
3. Repeat Step 2 to create an H-tree of order **2**, **3**, . . . , and so on, as shown in Figures 18.1c and d.

Write a program that draws an H-tree, as shown in Figure 18.1.

18.36 (*Sierpinski triangle*) Write a program that lets the user to enter the order and display the filled Sierpinski triangles as shown in Figure 18.18.



Enter an order: 0

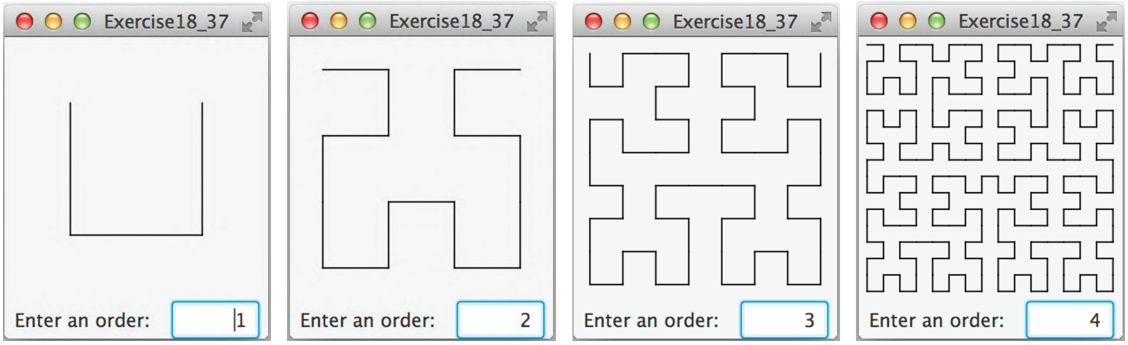
Enter an order: 1

Enter an order: 2

Enter an order: 3

FIGURE 18.18 A filled Sierpinski triangle is displayed. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****18.37** (*Hilbert curve*) The Hilbert curve, first described by German mathematician David Hilbert in 1891, is a space-filling curve that visits every point in a square grid with a size of 2 × 2, 4 × 4, 8 × 8, 16 × 16, or any other power of 2. Write a program that displays a Hilbert curve for the specified order, as shown in Figure 18.19.



Enter an order: 1

Enter an order: 2

Enter an order: 3

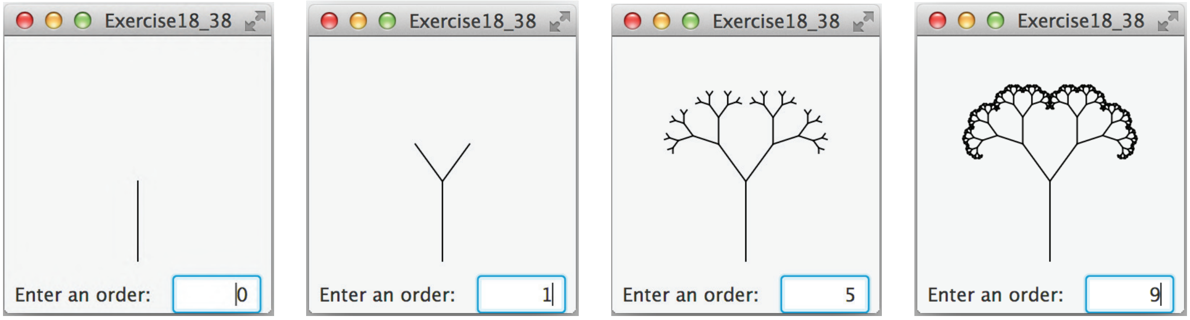
Enter an order: 4

(a) (b) (c) (d)

FIGURE 18.19 A Hilbert curve with the specified order is drawn. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

VideoNote
Recursive tree

****18.38** (*Recursive tree*) Write a program to display a recursive tree as shown in Figure 18.20.


Exercise18_38
Exercise18_38
Exercise18_38
Exercise18_38

Enter an order: 0

(a)

Enter an order: 1

(b)

Enter an order: 5

(c)

Enter an order: 9

(d)

FIGURE 18.20 A recursive tree with the specified depth is drawn. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****18.39** (*Drag the tree*) Revise Programming Exercise 18.38 to move the tree to where the mouse is dragged.

CHAPTER

19

GENERICS

Objectives

- To describe the benefits of generics (§19.2).
- To use generic classes and interfaces (§19.2).
- To define generic classes and interfaces (§19.3).
- To explain why generic types can improve reliability and readability (§19.3).
- To define and use generic methods and bounded generic types (§19.4).
- To develop a generic sort method to sort an array of **`Comparable`** objects (§19.5).
- To use raw types for backward compatibility (§19.6).
- To explain why wildcard generic types are necessary (§19.7).
- To describe generic-type erasure and list certain restrictions and limitations on generic types caused by type erasure (§19.8).
- To design and implement generic matrix classes (§19.9).

19.1 Introduction

Generics enable you to detect errors at compile time rather than at runtime.

Key Point

what is generics?

You have used a generic class `ArrayList` in Chapter 11, and generic interface `Comparable` in Chapter 13. *Generics* let you parameterize types. With this capability, you can define a class or a method with generic types that the compiler can replace with concrete types. For example, Java defines a generic `ArrayList` class for storing the elements of a generic type. From this generic class, you can create an `ArrayList` object for holding strings, and an `ArrayList` object for holding numbers. Here, strings and numbers are concrete types that replace the generic type.

why generics?

The key benefit of generics is to enable errors to be detected at compile time rather than at runtime. A generic class or method permits you to specify allowable types of objects that the class or method can work with. If you attempt to use an incompatible object, the compiler will detect that error.

This chapter explains how to define and use generic classes, interfaces, and methods and demonstrates how generics can be used to improve software reliability and readability. It can be intertwined with Chapter 13, Abstract Classes and Interfaces.

19.2 Motivations and Benefits

The motivation for using Java generics is to detect errors at compile time.

Key Point

Java has allowed you to define generic classes, interfaces, and methods since JDK 1.5. Several interfaces and classes in the Java API were modified using generics. For example, prior to JDK 1.5, the `java.lang.Comparable` interface was defined as shown in Figure 19.1a, but since JDK 1.5, it has been modified as shown in Figure 19.1b.

```
package java.lang;

public interface Comparable {
  public int compareTo(Object o)
}
```

(a) Prior to JDK 1.5

```
package java.lang;

public interface Comparable<T> {
  public int compareTo(T o)
}
```

(b) JDK 1.5

FIGURE 19.1 The `java.lang.Comparable` interface was modified in JDK 1.5 with a generic type.

formal generic type
actual concrete type
generic instantiation

Here, `<T>` represents a *formal generic type*, which can be replaced later with an *actual concrete type*. Replacing a generic type is called a *generic instantiation*. By convention, a single capital letter such as `E` or `T` is used to denote a formal generic type.

To see the benefits of using generics, let us examine the code in Figure 19.2. The statement in Figure 19.2a declares that `c` is a reference variable whose type is `Comparable` and invokes the `compareTo` method to compare a `Date` object with a string. The code compiles fine, but it has a runtime error because a string cannot be compared with a date.

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a) Prior to JDK 1.5

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

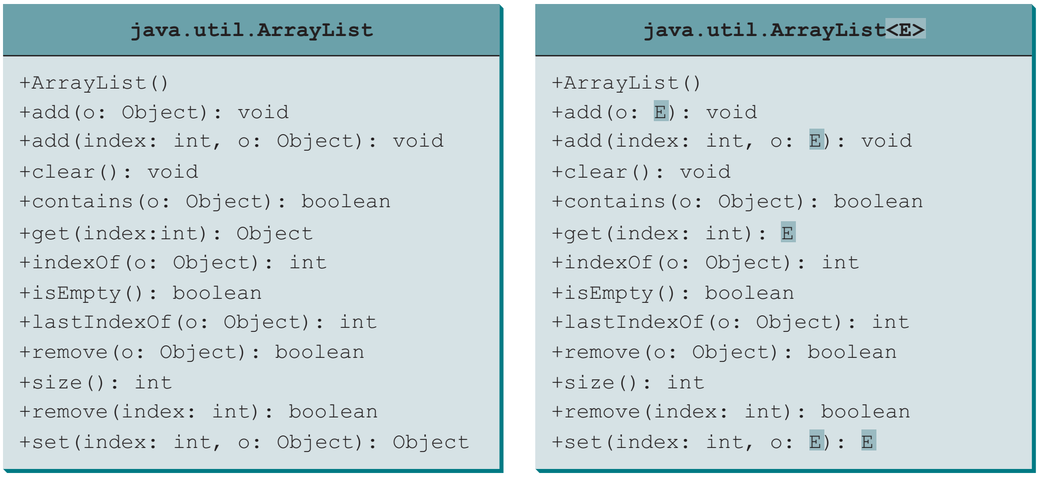
(b) JDK 1.5

FIGURE 19.2 The new generic type detects possible errors at compile time.

The statement in Figure 19.2b declares that `c` is a reference variable whose type is `Comparable<Date>` and invokes the `compareTo` method to compare a `Date` object with a string. This code has a compile error because the argument passed to the `compareTo` method must be of the **`Date`** type. Since the errors can be detected at compile time rather than at runtime, the generic type makes the program more reliable.

reliable

`ArrayList` was introduced in Section 11.11, The **`ArrayList`** Class. This class has been a generic class since JDK 1.5. Figure 19.3 shows the class diagram for **`ArrayList`** before and since JDK 1.5, respectively.


java.util.ArrayList
+ArrayList()
+add(o: Object): void
+add(index: int, o: Object): void
+clear(): void
+contains(o: Object): boolean
+get(index:int): Object
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: Object): Object
java.util.ArrayList<E>
+ArrayList()
+add(o: E): void
+add(index: int, o: E): void
+clear(): void
+contains(o: Object): boolean
+get(index: int): E
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: E): E

(a) `ArrayList` before JDK 1.5 (b) `ArrayList` since JDK 1.5

FIGURE 19.3 **`ArrayList`** is a generic class since JDK 1.5.

For example, the following statement creates a list for strings:

```
ArrayList<String> list = new ArrayList<>();
```

You can now add *only strings* into the list. For instance,

only strings allowed

```
list.add("Red");
```

If you attempt to add a nonstring, a compile error will occur. For example, the following statement is now illegal because **`list`** can contain only strings.

```
list.add(Integer.valueOf(1));
```

Generic types must be reference types. You cannot replace a generic type with a primitive type such as **`int`**, **`double`**, or **`char`**. For example, the following statement is wrong:

generic reference type

```
ArrayList<int> intList = new ArrayList<>();
```

To create an **`ArrayList`** object for **`int`** values, you have to use

```
ArrayList<Integer> intList = new ArrayList<>();
```

You can add an **`int`** value to **`intList`**. For example,

```
intList.add(5);
```

Java automatically wraps **`5`** into an **`Integer`** object. This is called *autoboxing*, as introduced in Section 10.8, Automatic Conversion between Primitive Types and Wrapper Class Types.

autoboxing

no casting needed

Casting is not needed to retrieve a value from a list with a specified element type because the compiler already knows the element type. For example, the following statements create a list that contains strings, add strings to the list, and retrieve strings from the list.

```
ArrayList<String> list = new ArrayList<>();
list.add("Red");
list.add("White");
String s = list.get(0); // No casting is needed
```

Prior to JDK 1.5, without using generics, you would have had to cast the return value to `String` as

```
String s = (String)(list.get(0)); // Casting needed prior to JDK 1.5
```

autounboxing

If the elements are of wrapper types, such as `Integer`, `Double`, and `Character`, you can directly assign an element to a primitive-type variable. This is called *autounboxing*, as introduced in Section 10.8. For example, see the following code:

```
1 ArrayList<Double> list = new ArrayList<>();
2 list.add(5.5); // 5.5 is automatically converted to a Double object
3 list.add(3.0); // 3.0 is automatically converted to a Double object
4 Double doubleObject = list.get(0); // No casting is needed
5 double d = list.get(1); // Automatically converted to double
```

In lines 2 and 3, `5.5` and `3.0` are automatically converted into `Double` objects and added to `list`. In line 4, the first element in `list` is assigned to a `Double` variable. No casting is necessary because `list` is declared for `Double` objects. In line 5, the second element in `list` is assigned to a `double` variable. The object in `list.get(1)` is automatically converted into a primitive-type value.

Check Point

19.2.1 Are there any compile errors in (a) and (b)?

```
ArrayList dates = new ArrayList();
dates.add(new Date());
dates.add(new String());
```

(a) Prior to JDK 1.5

```
ArrayList<Date> dates =
  new ArrayList<>();
dates.add(new Date());
dates.add(new String());
```

(b) Since JDK 1.5

19.2.2 What is wrong in (a)? Is the code in (b) correct?

```
ArrayList dates = new ArrayList();
dates.add(new Date());
Date date = dates.get(0);
```

(a) Prior to JDK 1.5

```
ArrayList<Date> dates =
  new ArrayList<>();
dates.add(new Date());
Date date = dates.get(0);
```

(b) Since JDK 1.5

19.2.3 What are the benefits of using generic types?

19.3 Defining Generic Classes and Interfaces

Key Point

A generic type can be defined for a class or interface. A concrete type must be specified when using the class to create an object or using the class or interface to declare a reference variable.

Let us revise the stack class in Section 11.13, Case Study: A Custom Stack Class, to generalize the element type with a generic type. The new stack class, named **`GenericStack`**, is shown in Figure 19.4 and is implemented in Listing 19.1.

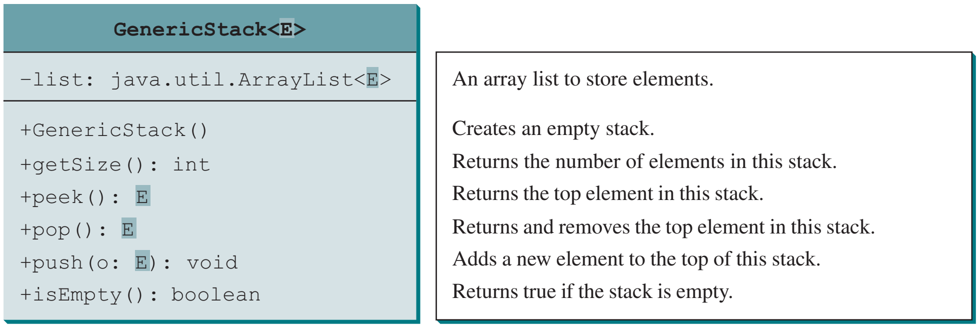

GenericStack<E>
-list: java.util.ArrayList<E>
+GenericStack()
+getSize(): int
+peek(): E
+pop(): E
+push(o: E): void
+isEmpty(): boolean
An array list to store elements.
Creates an empty stack.
Returns the number of elements in this stack.
Returns the top element in this stack.
Returns and removes the top element in this stack.
Adds a new element to the top of this stack.
Returns true if the stack is empty.

FIGURE 19.4 The **`GenericStack`** class encapsulates the stack storage and provides the operations for manipulating the stack.

LISTING 19.1 GenericStack.java

```
public class GenericStack<E> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<>();

  public int getSize() {
    return list.size();
  }

  public E peek() {
    return list.get(getSize() - 1);
  }

  public void push(E o) {
    list.add(o);
  }

  public E pop() {
    E o = list.get(getSize() - 1);
    list.remove(getSize() - 1);
    return o;
  }

  public boolean isEmpty() {
    return list.isEmpty();
  }

  @Override
  public String toString() {
    return "stack: " + list.toString();
  }
}
```

generic type E declared
generic array list
`getSize`
`peek`
`push`
`pop`
`isEmpty`

The following example creates a stack to hold strings and adds three strings to the stack:

```
GenericStack<String> stack1 = new GenericStack<>();
stack1.push("London");
stack1.push("Paris");
stack1.push("Berlin");
```

This example creates a stack to hold integers and adds three integers to the stack:

```
GenericStack<Integer> stack2 = new GenericStack<>();
stack2.push(1); // autoboxing 1 to an Integer object
stack2.push(2);
stack2.push(3);
```

benefits of using generic types

Instead of using a generic type, you could simply make the type element `Object`, which can accommodate any object type. However, using a specific concrete type can improve software reliability and readability because certain errors can be detected at compile time rather than at runtime. For example, because `stack1` is declared `GenericStack<String>`, only strings can be added to the stack. It would be a compile error if you attempted to add an integer to `stack1`.

generic class constructor

Caution

To create a stack of strings, you use `new GenericStack<String>()` or `new GenericStack<>()`. This could mislead you into thinking that the constructor of `GenericStack` should be defined as

```
public GenericStack<E>()
```

This is wrong. It should be defined as

```
public GenericStack()
```

multiple generic parameters

Note

Occasionally, a generic class may have more than one parameter. In this case, place the parameters together inside the brackets, separated by commas—for example, `<E1, E2, E3>`.

inheritance with generics

Note

You can define a class or an interface as a subtype of a generic class or interface. For example, the `java.lang.String` class is defined to implement the `Comparable` interface in the Java API as follows:

```
public class String implements Comparable<String>
```

Check Point

19.3.1 What is the generic definition for `java.lang.Comparable` in the Java API?

19.3.2 Since you create an instance of `ArrayList` of strings using `new ArrayList<String>()`, should the constructor in the `ArrayList` class be defined as

```
public ArrayList<E>()
```

19.3.3 Can a generic class have multiple generic parameters?

19.3.4 How do you declare a generic type in a class?

19.4 Generic Methods

Key Point

A generic type can be defined for a static method.

generic method

You can define generic interfaces (e.g., the `Comparable` interface in Figure 19.1b) and classes (e.g., the `GenericStack` class in Listing 19.1). You can also use generic types to define generic methods. For example, Listing 19.2 defines a generic method `print` (lines 10–14) to print an array of objects. Line 6 passes an array of integer objects to invoke the generic `print` method. Line 7 invokes `print` with an array of strings.

LISTING 19.2 GenericMethodDemo.java

```
public class GenericMethodDemo {
  public static void main(String[] args ) {
    Integer[] integers = {1, 2, 3, 4, 5};
    String[] strings = {"London", "Paris", "New York", "Austin"};

    GenericMethodDemo.<Integer>print(integers);
    GenericMethodDemo.<String>print(strings);
  }

  public static <E> void print(E[] list) {
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
    System.out.println();
  }
}
```

generic method

declare a generic method

To declare a generic method, you place the generic type **`<E>`** immediately after the keyword **`static`** in the method header. For example,

```
public static <E> void print(E[] list)
```

invoke generic method

To invoke a generic method, prefix the method name with the actual type in angle brackets. For example,

```
GenericMethodDemo.<Integer>print(integers);
GenericMethodDemo.<String>print(strings);
```

or simply invoke it as follows:

```
print(integers);
print(strings);
```

In the latter case, the actual type is not explicitly specified. The compiler automatically discovers the actual type.

bounded generic type

A generic type can be specified as a subtype of another type. Such a generic type is called *bounded*. For example, Listing 19.3 revises the **`equalArea`** method in Listing 13.4, TestGeometricObject.java, to test whether two geometric objects have the same area. The bounded generic type **`<E extends GeometricObject>`** (line 10) specifies that **`E`** is a generic subtype of **`GeometricObject`**. You must invoke **`equalArea`** by passing two instances of **`GeometricObject`**.

LISTING 19.3 BoundedTypeDemo.java

```
 1  public class BoundedTypeDemo {
 2    public static void main(String[] args ) {
 3      Rectangle rectangle = new Rectangle(2, 2);
 4      Circle circle = new Circle(2);
 5
 6      System.out.println("Same area? " +
 7        equalArea(rectangle, circle));
 8    }
 9
10    public static <E extends GeometricObject> boolean equalArea(
11        E object1, E object2) {
12      return object1.getArea() == object2.getArea();
13    }
14  }
```

Rectangle in Listing 13.3
Circle in Listing 13.2

bounded generic type

Note
An unbounded generic type `<E>` is the same as `<E extends Object>`.

generic class parameter vs. generic method parameter

Note
To define a generic type for a class, place it after the class name, such as `GenericStack<E>`. To define a generic type for a method, place the generic type before the method return type, such as `<E> void max(E o1, E o2)`.

Check Point

19.4.1 How do you declare a generic method? How do you invoke a generic method?

19.4.2 What is a bounded generic type?

19.5 Case Study: Sorting an Array of Objects

Key Point

You can develop a generic method for sorting an array of `Comparable` *objects.*

This section presents a generic method for sorting an array of `Comparable` objects. The objects are instances of the `Comparable` interface and they are compared using the `compareTo` method. To test the method, the program sorts an array of integers, an array of double numbers, an array of characters, and an array of strings. The program is shown in Listing 19.4.

Listing 19.4 GenericSort.java

```java
public class GenericSort {
  public static void main(String[] args) {
    // Create an Integer array
    Integer[] intArray = {Integer.valueOf(2), Integer.valueOf(4),
      Integer.valueOf(3)};

    // Create a Double array
    Double[] doubleArray = {Double.valueOf(3.4), Double.valueOf(1.3),
      Double.valueOf(-22.1)};

    // Create a Character array
    Character[] charArray = {Character.valueOf('a'),
      Character.valueOf('J'), Character.valueOf('r')};

    // Create a String array
    String[] stringArray = {"Tom", "Susan", "Kim"};

    // Sort the arrays
    sort(intArray);
    sort(doubleArray);
    sort(charArray);
    sort(stringArray);

    // Display the sorted arrays
    System.out.print("Sorted Integer objects: ");
    printList(intArray);
    System.out.print("Sorted Double objects: ");
    printList(doubleArray);
    System.out.print("Sorted Character objects: ");
    printList(charArray);
    System.out.print("Sorted String objects: ");
    printList(stringArray);
  }

```

sort `Integer` objects
sort `Double` objects
sort `Character` objects
sort `String` objects

```java
35    /** Sort an array of comparable objects */
36    public static <E extends Comparable<E>> void sort(E[] list) {
37      E currentMin;
38      int currentMinIndex;
39
40      for (int i = 0; i < list.length - 1; i++) {
41        // Find the minimum in the list[i+1..list.length-2]
42        currentMin = list[i];
43        currentMinIndex = i;
44
45        for (int j = i + 1; j < list.length; j++) {
46          if (currentMin.compareTo(list[j]) > 0) {
47            currentMin = list[j];
48            currentMinIndex = j;
49          }
50        }
51
52        // Swap list[i] with list[currentMinIndex] if necessary;
53        if (currentMinIndex != i) {
54          list[currentMinIndex] = list[i];
55          list[i] = currentMin;
56        }
57      }
58    }
59
60    /** Print an array of objects */
61    public static void printList(Object[] list) {
62      for (int i = 0; i < list.length; i++)
63        System.out.print(list[i] + " ");
64      System.out.println();
65    }
66  }
```

generic sort method

`compareTo`

```
Sorted Integer objects: 2 3 4
Sorted Double objects: -22.1 1.3 3.4
Sorted Character objects: J a r
Sorted String objects: Kim Susan Tom
```

The algorithm for the `sort` method is the same as in Listing 7.8, SelectionSort.java. The `sort` method in that program sorts an array of `double` values. The `sort` method in this example can sort an array of any object type, provided that the objects are also instances of the `Comparable` interface. The generic type is defined as `<E extends Comparable<E>>` (line 36). This has two meanings. First, it specifies that `E` is a subtype of `Comparable`. Second, it specifies that the elements to be compared are of the `E` type as well.

The `sort` method uses the `compareTo` method to determine the order of the objects in the array (line 46). `Integer`, `Double`, `Character`, and `String` implement `Comparable`, so the objects of these classes can be compared using the `compareTo` method. The program creates arrays of `Integer` objects, `Double` objects, `Character` objects, and `String` objects (lines 4–16) and invokes the `sort` method to sort these arrays (lines 19–22).

Check Point

19.5.1 Given `int[] list = {1, 2, -1}`, can you invoke `sort(list)` using the `sort` method in Listing 19.4?

19.5.2 Given `int[] list = {Integer.valueOf(1), Integer.valueOf(2), Integer.valueOf(-1)}`, can you invoke `sort(list)` using the `sort` method in Listing 19.4?

19.6 Raw Types and Backward Compatibility

Key Point

A generic class or interface used without specifying a concrete type, called a raw type, enables backward compatibility with earlier versions of Java.

You can use a generic class without specifying a concrete type such as the following:

```
GenericStack stack = new GenericStack(); // raw type
```

This is roughly equivalent to

```
GenericStack<Object> stack = new GenericStack<Object>();
```

raw type
backward compatibility

A generic class such as `GenericStack` and `ArrayList` used without a type parameter is called a *raw type*. Using raw types allows for backward compatibility with earlier versions of Java. For example, a generic type has been used in `java.lang.Comparable` since JDK 1.5, but a lot of code still uses the raw type `Comparable`, as given in Listing 19.5:

Listing 19.5 `Max.java`

raw type

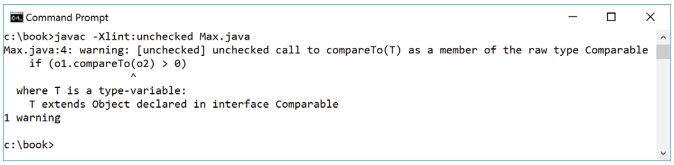
```
1  public class Max {
2    /** Return the maximum of two objects */
3    public static Comparable max(Comparable o1, Comparable o2) {
4      if (o1.compareTo(o2) > 0)
5        return o1;
6      else
7        return o2;
8    }
9  }
```

`Comparable o1` and `Comparable o2` are raw type declarations. Be careful: *raw types are unsafe*. For example, you might invoke the `max` method using

```
Max.max("Welcome", 23); // 23 is autoboxed into an Integer object
```

-Xlint:unchecked

This would cause a runtime error because you cannot compare a string with an integer object. The Java compiler displays a warning on line 3 when compiled with the option ***`-Xlint:unchecked`***, as shown in Figure 19.5.



```
c:\book>javac -Xlint:unchecked Max.java
Max.java:4: warning: [unchecked] unchecked call to compareTo(T) as a member of the raw type Comparable
    if (o1.compareTo(o2) > 0)
                    ^
  where T is a type-variable:
    T extends Object declared in interface Comparable
1 warning

c:\book>
```

Figure 19.5 The unchecked warnings are displayed using the compiler option `-Xlint:unchecked`. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

A better way to write the `max` method is to use a generic type, as given in Listing 19.6.

Listing 19.6 `MaxUsingGenericType.java`

bounded type

```
1  public class MaxUsingGenericType {
2    /** Return the maximum of two objects */
3    public static <E  extends Comparable<E>> E max(E o1, E o2) {
4      if (o1.compareTo(o2) > 0)
5        return o1;
```

```
    else
      return o2;
  }
}
```

If you invoke the `max` method using

```
// 23 is autoboxed into an Integer object
MaxUsingGenericType.max("Welcome", 23);
```

a compile error will be displayed because the two arguments of the `max` method in `MaxUsingGenericType` must have the same type (e.g., two strings or two integer objects). Furthermore, the type `E` must be a subtype of `Comparable<E>`.

As another example, in the following code you can declare a raw type `stack` in line 1, assign `new GenericStack<String>` to it in line 2, and push a string and an integer object to the stack in lines 3 and 4:

```
1  GenericStack stack;
2  stack = new GenericStack<String>();
3  stack.push("Welcome to Java");
4  stack.push(Integer.valueOf(2));
```

However, line 4 is unsafe because the stack is intended to store strings, but an `Integer` object is added into the stack. Line 3 should be okay, but the compiler will show warnings for both line 3 and line 4, because it cannot follow the semantic meaning of the program. All the compiler knows is that stack is a raw type, and performing certain operations is unsafe. Therefore, warnings are displayed to alert potential problems.

Tip

Since raw types are unsafe, this book will not use them from here on.

19.6.1 What is a raw type? Why is a raw type unsafe? Why is the raw type allowed in Java?

19.6.2 What is the syntax to declare an `ArrayList` reference variable using the raw type and assign a raw type `ArrayList` object to it?

Check Point

19.7 Wildcard Generic Types

You can use unbounded wildcards, bounded wildcards, or lower bound wildcards to specify a range for a generic type.

Key Point

What are wildcard generic types, and why are they needed? Listing 19.7 gives an example to demonstrate the needs. The example defines a generic `max` method for finding the maximum in a stack of numbers (lines 12–22). The main method creates a stack of integer objects, adds three integers to the stack, and invokes the `max` method to find the maximum number in the stack.

LISTING 19.7 WildCardNeedDemo.java

```
 1  public class WildCardNeedDemo {
 2    public static void main(String[] args ) {
 3      GenericStack<Integer> intStack = new GenericStack<>();
 4      intStack.push(1); // 1 is autoboxed into an Integer object
 5      intStack.push(2);
 6      intStack.push(-2);
 7
 8      System.out.print("The max number is " + max(intStack));
 9    }
10
```

`GenericStack<Integer>` type

GenericStack<Number> type

```
11   /** Find the maximum in a stack of numbers */
12   public static double max(GenericStack<Number> stack) {
13     double max = stack.pop().doubleValue(); // Initialize max
14
15     while (!stack.isEmpty()) {
16       double value = stack.pop().doubleValue();
17       if (value > max)
18         max = value;
19     }
20
21     return max;
22   }
23 }
```

The program in Listing 19.7 has a compile error in line 8 because `intStack` is not an instance of `GenericStack<Number>`. Thus, you cannot invoke `max(intStack)`.

The fact is `Integer` is a subtype of `Number`, but `GenericStack<Integer>` is not a subtype of `GenericStack<Number>`. To circumvent this problem, use wildcard generic types. A wildcard generic type has three forms: `?`, `? extends T`, and `? super T`, where `T` is a generic type.

unbounded wildcard

bounded wildcard

lower bound wildcard

The first form, `?`, called an *unbounded wildcard*, is the same as `? extends Object`. The second form, `? extends T`, called a *bounded wildcard*, represents `T` or a subtype of `T`. The third form, `? super T`, called a *lower bound wildcard*, denotes `T` or a supertype of `T`.

You can fix the error by replacing line 12 in Listing 19.7 as follows:

```
public static double max(GenericStack<? extends Number> stack) {
```

`<? extends Number>` is a wildcard type that represents `Number` or a subtype of `Number`, so it is legal to invoke `max(new GenericStack<Integer>())` or `max(new GenericStack<Double>())`.

Listing 19.8 shows an example of using the `?` wildcard in the `print` method that prints objects in a stack and empties the stack. `<?>` is a wildcard that represents any object type. It is equivalent to `<? extends Object>`. What happens if you replace `GenericStack<?>` with `GenericStack<Object>`? It would be wrong to invoke `print(intStack)` because `intStack` is not an instance of `GenericStack<Object>`. Note that `GenericStack<Integer>` is not a subtype of `GenericStack<Object>` even though `Integer` is a subtype of `Object`.

LISTING 19.8 AnyWildCardDemo.java

GenericStack<Integer> type

wildcard type

```
public class AnyWildCardDemo {
  public static void main(String[] args) {
    GenericStack<Integer> intStack = new GenericStack<>();
    intStack.push(1); // 1 is autoboxed into an Integer object
    intStack.push(2);
    intStack.push(-2);

    print(intStack);
  }

  /** Prints objects and empties the stack */
  public static void print(GenericStack<?> stack) {
    while (!stack.isEmpty()) {
      System.out.print(stack.pop() + " ");
    }
  }
}
```

why `<? Super T>`

When is the wildcard `<? super T>` needed? Consider the example in Listing 19.9. The example creates a stack of strings in `stack1` (line 3) and a stack of objects in `stack2` (line 4) and invokes `add(stack1, stack2)` (line 8) to add the strings in `stack1` into `stack2`. `GenericStack<? super T>` is used to declare `stack2` in line 13. If `<? super T>` is replaced by `<T>`, a compile error will occur on `add(stack1, stack2)` in line 8 because `stack1`'s type is `GenericStack<String>` and `stack2`'s type is `GenericStack<Object>`. `<? super T>` represents type `T` or a supertype of `T`. `Object` is a supertype of `String`.

LISTING 19.9 SuperWildCardDemo.java

GenericStack<String> type

<? Super T> type

```
 1  public class SuperWildCardDemo {
 2    public static void main(String[] args) {
 3      GenericStack<String> stack1 = new GenericStack<>();
 4      GenericStack<Object> stack2 = new GenericStack<>();
 5      stack2.push("Java");
 6      stack2.push(2);
 7      stack1.push("Sun");
 8      add(stack1, stack2);
 9      AnyWildCardDemo.print(stack2);
10    }
11
12    public static <T> void add(GenericStack<T> stack1,
13        GenericStack<? super T> stack2) {
14      while (!stack1.isEmpty())
15        stack2.push(stack1.pop());
16    }
17  }
```

This program will also work if the method header in lines 12 and 13 is modified as follows:

```
public static <T> void add(GenericStack<? extends T> stack1,
    GenericStack<T> stack2)
```

The inheritance relationship involving generic types and wildcard types is summarized in Figure 19.6. In this figure, `A` and `B` represent classes or interfaces, and `E` is a generic-type parameter.

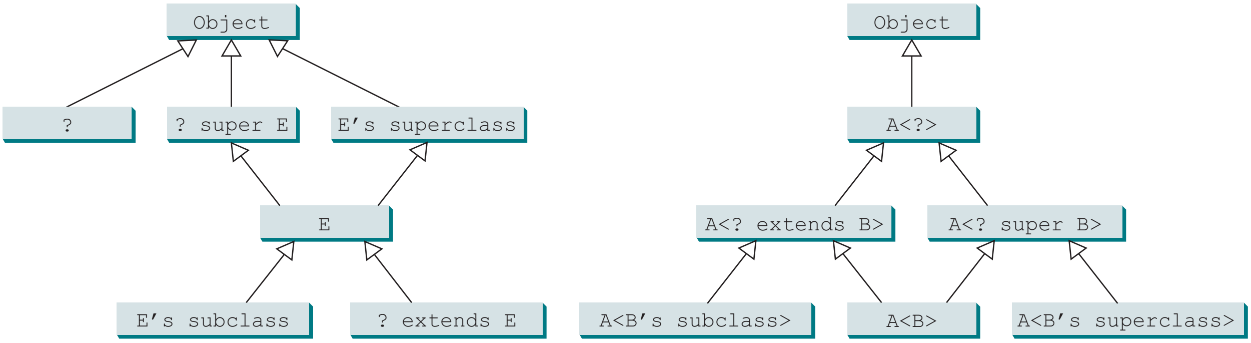


FIGURE 19.6 The relationship between generic types and wildcard types.

Check Point

19.7.1 Is `GenericStack` the same as `GenericStack<Object>`?

19.7.2 What is an unbounded wildcard, a bounded wildcard, and a lower bound wildcard?

19.7.3 What happens if lines 12 and 13 in Listing 19.9 are changed to

```
public static <T> void add(GenericStack<T> stack1,
    GenericStack<T> stack2)
```

19.7.4 What happens if lines 12 and 13 in Listing 19.9 are changed to

```
public static <T> void add(GenericStack<? extends T> stack1,
    GenericStack<T> stack2)
```

19.8 Erasure and Restrictions on Generics

Key Point

The information on generics is used by the compiler but is not available at runtime. This is called type erasure.

type erasure

Generics are implemented using an approach called *type erasure*: The compiler uses the generic-type information to compile the code, but erases it afterward. Thus, the generic information is not available at runtime. This approach enables the generic code to be backward compatible with the legacy code that uses raw types.

erase generics

The generics are present at compile time. Once the compiler confirms that a generic type is used safely, it converts the generic type to a raw type. For example, the compiler checks whether the following code in (a) uses generics correctly, then translates it into the equivalent code in (b) for runtime use. The code in (b) uses the raw type.

```
ArrayList<String> list = new ArrayList<>();
list.add("Oklahoma");
String state = list.get(0);
```

(a)

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

(b)

replace generic type

When generic classes, interfaces, and methods are compiled, the compiler replaces the generic type with the `Object` type. For example, the compiler would convert the following method in (a) into (b).

```
public static <E> void print(E[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(a)

```
public static void print(Object[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(b)

replace bounded type

If a generic type is bounded, the compiler replaces it with the bounded type. For example, the compiler would convert the following method in (a) into (b).

```
public static <E extends GeometricObject>
    boolean equalArea(
      E object1,
      E object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(a)

```
public static
    boolean equalArea(
      GeometricObject object1,
      GeometricObject object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(b)

important fact

It is important to note a generic class is shared by all its instances regardless of its actual concrete type. Suppose `list1` and `list2` are created as follows:

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<Integer> list2 = new ArrayList<>();
```

Although `ArrayList<String>` and `ArrayList<Integer>` are two types at compile time, only one `ArrayList` class is loaded into the JVM at runtime. `list1` and `list2` are both instances of `ArrayList`, so the following statements display `true`:

```
System.out.println(list1 instanceof ArrayList);
System.out.println(list2 instanceof ArrayList);
```

However, the expression `list1 instanceof ArrayList<String>` is wrong. Since `ArrayList<String>` is not stored as a separate class in the JVM, using it at runtime makes no sense.

Because generic types are erased at runtime, there are certain restrictions on how generic types can be used. Here are some of the restrictions:

Restriction 1: Cannot Use `new E()`

You cannot create an instance using a generic-type parameter. For example, the following statement is wrong:

```
E object = new E();
```

`no new E()`

The reason is `new E()` is executed at runtime, but the generic type `E` is not available at runtime.

Restriction 2: Cannot Use `new E[]`

You cannot create an array using a generic type parameter. For example, the following statement is wrong:

```
E[] elements = new E[capacity];
```

`no new E[capacity]`

You can circumvent this limitation by creating an array of the `Object` type then casting it to `E[]`, as follows:

```
E[] elements = (E[])new Object[capacity];
```

unavoidable compile warning

However, casting to `(E[])` causes an unchecked compile warning. The warning occurs because the compiler is not certain that casting will succeed at runtime. For example, if `E` is `String` and `new Object[]` is an array of `Integer` objects, `(String[])(new Object[])` will cause a `ClassCastException`. This type of compile warning is a limitation of Java generics and is unavoidable.

Generic array creation using a generic class is not allowed, either. For example, the following code is wrong:

```
ArrayList<String>[] list = new ArrayList<String>[10];
```

You can use the following code to circumvent this restriction:

```
ArrayList<String>[] list = (ArrayList<String>[])new
  ArrayList[10];
```

However, you will still get a compile warning.

Restriction 3: A Generic Type Parameter of a Class Is Not Allowed in a Static Context

Since all instances of a generic class have the same runtime class, the static variables and methods of a generic class are shared by all its instances. Therefore, it is illegal to refer to a generic-type parameter for a class in a static method, field, or initializer. For example, the following code is illegal:

```
public class Test<E> {
  public static void m(E o1) {  // Illegal
  }
```

```
    public static E o1; // Illegal

    static {
      E o2; // Illegal
    }
  }
```

Restriction 4: Exception Classes Cannot Be Generic

A generic class may not extend `java.lang.Throwable`, so the following class declaration would be illegal:

```
public class MyException<T> extends Exception {
}
```

Why? If it were allowed, you would have a `catch` clause for `MyException<T>` as follows:

```
try {
  ...
}
catch (MyException<T> ex) {
  ...
 }
```

The JVM has to check the exception thrown from the `try` clause to see if it matches the type specified in a `catch` clause. This is impossible, because the type information is not present at runtime.

Check Point

19.8.1 What is erasure? Why are Java generics implemented using erasure?

19.8.2 If your program uses `ArrayList<String>` and `ArrayList<Date>`, does the JVM load both of them?

19.8.3 Can you create an instance using `new E()` for a generic type `E`? Why?

19.8.4 Can a method that uses a generic class parameter be static? Why?

19.8.5 Can you define a custom generic exception class? Why?

19.9 Case Study: Generic Matrix Class

Key Point

This section presents a case study on designing classes for matrix operations using generic types.

The addition and multiplication operations for all matrices are similar except that their element types differ. Therefore, you can design a superclass that describes the common operations shared by matrices of all types regardless of their element types, and you can define subclasses tailored to specific types of matrices. This case study gives implementations for two types: `int` and `Rational`. For the `int` type, the wrapper class `Integer` should be used to wrap an `int` value into an object, so the object is passed in the methods for operations.

The class diagram is shown in Figure 19.7. The methods `addMatrix` and `multiplyMatrix` add and multiply two matrices of a generic type `E[][]`. The static method `printResult` displays the matrices, the operator, and their result. The methods `add`, `multiply`, and `zero` are abstract because their implementations depend on the specific type of the array elements. For example, the `zero()` method returns `0` for the `Integer` type and `0/1` for the `Rational` type. These methods will be implemented in the subclasses in which the matrix element type is specified.

`IntegerMatrix` and `RationalMatrix` are concrete subclasses of `GenericMatrix`. These two classes implement the `add`, `multiply`, and `zero` methods defined in the `GenericMatrix` class.

Listing 19.10 implements the `GenericMatrix` class. `<E extends Number>` in line 1 specifies the generic type is a subtype of `Number`. Three abstract methods—`add`, `multiply`,

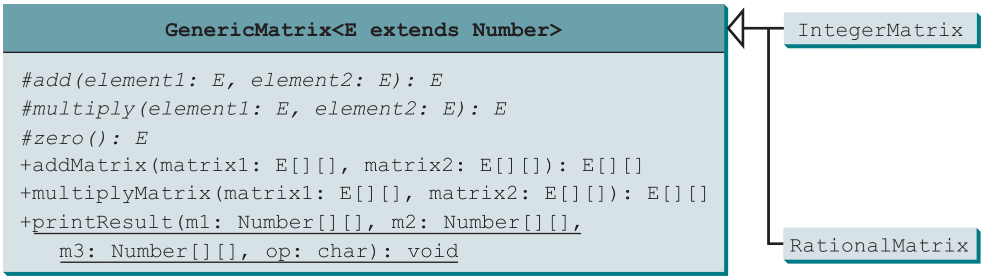

GenericMatrix<E extends Number>
#add(element1: E, element2: E): E
#multiply(element1: E, element2: E): E
#zero(): E
+addMatrix(matrix1: E[][], matrix2: E[][]): E[][]
+multiplyMatrix(matrix1: E[][], matrix2: E[][]): E[][]
+printResult(m1: Number[][], m2: Number[][], m3: Number[][], op: char): void
IntegerMatrix
RationalMatrix

FIGURE 19.7 The `GenericMatrix` class is an abstract superclass for `IntegerMatrix` and `RationalMatrix`.

and `zero`—are defined in lines 3, 6, and 9. These methods are abstract because we cannot implement them without knowing the exact type of the elements. The `addMaxtrix` (lines 12–30) and `multiplyMatrix` (lines 33–57) methods implement the methods for adding and multiplying two matrices. All these methods must be nonstatic because they use generic-type `E` for the class. The `printResult` method (lines 60–84) is static because it is not tied to specific instances.

The matrix element type is a generic subtype of `Number`. This enables you to use an object of any subclass of `Number` as long as you can implement the abstract `add`, `multiply`, and `zero` methods in subclasses.

The `addMatrix` and `multiplyMatrix` methods (lines 12–57) are concrete methods. They are ready to use as long as the `add`, `multiply`, and `zero` methods are implemented in the subclasses.

The `addMatrix` and `multiplyMatrix` methods check the bounds of the matrices before performing operations. If the two matrices have incompatible bounds, the program throws an exception (lines 16 and 36).

LISTING 19.10 GenericMatrix.java

```
 1 public abstract class GenericMatrix<E extends Number> {
 2   /** Abstract method for adding two elements of the matrices */
 3   protected abstract E add(E o1, E o2);
 4
 5   /** Abstract method for multiplying two elements of the matrices */
 6   protected abstract E multiply(E o1, E o2);
 7
 8   /** Abstract method for defining zero for the matrix element */
 9   protected abstract E zero();
10
11   /** Add two matrices */
12   public E[][] addMatrix(E[][] matrix1, E[][] matrix2) {
13     // Check bounds of the two matrices
14     if ((matrix1.length != matrix2.length) ||
15         (matrix1[0].length != matrix2[0].length)) {
16       throw new RuntimeException(
17         "The matrices do not have the same size");
18     }
19
20     E[][] result =
21       (E[][])new Number[matrix1.length][matrix1[0].length];
22
23     // Perform addition
24     for (int i = 0; i < result.length; i++)
25       for (int j = 0; j < result[i].length; j++) {
26         result[i][j] = add(matrix1[i][j], matrix2[i][j]);
27       }
28
```

bounded generic type

abstract method

abstract method

abstract method

add two matrices

```
    return result;
  }

  /** Multiply two matrices */
```

multiply two matrices

```
  public E[][] multiplyMatrix(E[][] matrix1, E[][] matrix2) {
    // Check bounds
    if (matrix1[0].length != matrix2.length) {
      throw new RuntimeException(
        "The matrices do not have compatible size");
    }

    // Create result matrix
    E[][] result =
      (E[][])new Number[matrix1.length][matrix2[0].length];

    // Perform multiplication of two matrices
    for (int i = 0; i < result.length; i++) {
      for (int j = 0; j < result[0].length; j++) {
        result[i][j] = zero();

        for (int k = 0; k < matrix1[0].length; k++) {
          result[i][j] = add(result[i][j],
            multiply(matrix1[i][k], matrix2[k][j]));
        }
      }
    }

    return result;
  }

  /** Print matrices, the operator, and their operation result */
```

display result

```
  public static void printResult(
      Number[][] m1, Number[][] m2, Number[][] m3, char op) {
    for (int i = 0; i < m1.length; i++) {
      for (int j = 0; j < m1[0].length; j++)
        System.out.print(" " + m1[i][j]);

      if (i == m1.length / 2)
        System.out.print("  " + op + "  ");
      else
        System.out.print("     ");

      for (int j = 0; j < m2.length; j++)
        System.out.print(" " + m2[i][j]);

      if (i == m1.length / 2)
        System.out.print("  =  ");
      else
        System.out.print("     ");

      for (int j = 0; j < m3.length; j++)
        System.out.print(m3[i][j] + " ");

      System.out.println();
    }
  }
}
```

Listing 19.11 implements the **`IntegerMatrix`** class. The class extends **`GenericMatrix<Integer>`** in line 1. After the generic instantiation, the **`add`** method in **`GenericMatrix<Integer>`**

is now `Integer add(Integer o1, Integer o2)`. The `add`, `multiply`, and `zero` methods are implemented for `Integer` objects. These methods are still protected because they are invoked only by the `addMatrix` and `multiplyMatrix` methods.

Design Pattern Note

The code in the `GenericMatrix` class applies the template method pattern, which implements a method using abstract methods whose concrete implementation will be provided in the subclasses. In the `GenericMatrix`, the `addMatrix` and `multiplyMatrix` methods are implemented using the abstract `add`, `multiply`, and `zero` methods whose concrete implementation will be provided in the subclasses `IntegerMatrix` and `RationalMatrix`.

LISTING 19.11 IntegerMatrix.java

```java
public class IntegerMatrix extends GenericMatrix<Integer> {
  @Override /** Add two integers */
  protected Integer add(Integer o1, Integer o2) {
    return o1 + o2;
  }

  @Override /** Multiply two integers */
  protected Integer multiply(Integer o1, Integer o2) {
    return o1 * o2;
  }

  @Override /** Specify zero for an integer */
  protected Integer zero() {
    return 0;
  }
}
```

extends generic type
implement add
implement `multiply`
implement `zero`

Listing 19.12 implements the `RationalMatrix` class. The `Rational` class was introduced in Listing 13.13, Rational.java. `Rational` is a subtype of `Number`. The `RationalMatrix` class extends `GenericMatrix<Rational>` in line 1. After the generic instantiation, the `add` method in `GenericMatrix<Rational>` is now `Rational add(Rational r1, Rational r2)`. The `add`, `multiply`, and `zero` methods are implemented for `Rational` objects. These methods are still protected because they are invoked only by the `addMatrix` and `multiplyMatrix` methods.

LISTING 19.12 RationalMatrix.java

```java
 1  public class RationalMatrix extends GenericMatrix<Rational> {
 2    @Override /** Add two rational numbers */
 3    protected Rational add(Rational r1, Rational r2) {
 4      return r1.add(r2);
 5    }
 6
 7    @Override /** Multiply two rational numbers */
 8    protected Rational multiply(Rational r1, Rational r2) {
 9      return r1.multiply(r2);
10    }
11
12    @Override /** Specify zero for a Rational number */
13    protected Rational zero() {
14      return new Rational(0, 1);
15    }
16  }
```

extends generic type
implement add
implement `multiply`
implement `zero`

Listing 19.13 gives a program that creates two **`Integer`** matrices (lines 4 and 5) and an **`IntegerMatrix`** object (line 8), and adds and multiplies two matrices in lines 12 and 16.

LISTING 19.13 TestIntegerMatrix.java

create matrices

create `IntegerMatrix`

add two matrices

multiply two matrices

```
 1 public class TestIntegerMatrix {
 2   public static void main(String[] args) {
 3     // Create Integer arrays m1, m2
 4     Integer[][] m1 = new Integer[][]{{1, 2, 3}, {4, 5, 6}, {1, 1, 1}};
 5     Integer[][] m2 = new Integer[][]{{1, 1, 1}, {2, 2, 2}, {0, 0, 0}};
 6
 7     // Create an instance of IntegerMatrix
 8     IntegerMatrix integerMatrix = new IntegerMatrix();
 9
10     System.out.println("\nm1 + m2 is ");
11     GenericMatrix.printResult(
12       m1, m2, integerMatrix.addMatrix(m1, m2), '+');
13
14     System.out.println("\nm1 * m2 is ");
15     GenericMatrix.printResult(
16       m1, m2, integerMatrix.multiplyMatrix(m1, m2), '*');
17   }
18 }
```

```
m1 + m2 is
 1 2 3     1 1 1     2 3 4
 4 5 6  +  2 2 2  =  6 7 8
 1 1 1     0 0 0     1 1 1

m1 * m2 is
 1 2 3     1 1 1     5  5  5
 4 5 6  *  2 2 2  =  14 14 14
 1 1 1     0 0 0     3  3  3
```

Listing 19.14 gives a program that creates two **`Rational`** matrices (lines 4–10) and a **`RationalMatrix`** object (line 13) and adds and multiplies two matrices in lines 17 and 19.

LISTING 19.14 TestRationalMatrix.java

create matrices

create `RationalMatrix`

add two matrices

```
 1 public class TestRationalMatrix {
 2   public static void main(String[] args) {
 3     // Create two Rational arrays m1 and m2
 4     Rational[][] m1 = new Rational[3][3];
 5     Rational[][] m2 = new Rational[3][3];
 6     for (int i = 0; i < m1.length; i++)
 7       for (int j = 0; j < m1[0].length; j++) {
 8         m1[i][j] = new Rational(i + 1, j + 5);
 9         m2[i][j] = new Rational(i + 1, j + 6);
10       }
11
12     // Create an instance of RationalMatrix
13     RationalMatrix rationalMatrix = new RationalMatrix();
14
15     System.out.println("\nm1 + m2 is ");
16     GenericMatrix.printResult(
17       m1, m2, rationalMatrix.addMatrix(m1, m2), '+');
18
19     System.out.println("\nm1 * m2 is ");
```

```
    GenericMatrix.printResult(
      m1, m2, rationalMatrix.multiplyMatrix(m1, m2), '*');
  }
}
```

multiply two matrices

```
m1 + m2 is
 1/5 1/6 1/7     1/6 1/7 1/8     11/30 13/42 15/56
 2/5 1/3 2/7  +  1/3 2/7 1/4  =  11/15 13/21 15/28
 3/5 1/2 3/7     1/2 3/7 3/8     11/10 13/14 45/56

m1 * m2 is
 1/5 1/6 1/7     1/6 1/7 1/8     101/630 101/735 101/840
 2/5 1/3 2/7  *  1/3 2/7 1/4  =  101/315 202/735 101/420
 3/5 1/2 3/7     1/2 3/7 3/8     101/210 101/245 101/280
```

Check Point

19.9.1 Why are the `add`, `multiple`, and `zero` methods defined abstract in the `GenericMatrix` class?

19.9.2 How are the `add`, `multiple`, and `zero` methods implemented in the `IntegerMatrix` class?

19.9.3 How are the `add`, `multiple`, and `zero` methods implemented in the `RationalMatrix` class?

19.9.4 What would be wrong if the `printResult` method is defined as follows?

```
public static void printResult(
  E[][] m1, E[][] m2, E[][] m3, char op)
```

KEY TERMS

actual concrete type	752
bounded generic type	757
bounded wildcard (`<? extends E>`)	762
formal generic type	752
generic instantiation	752
lower bound wildcard (`<? super E>`)	762
raw type	760
unbounded wildcard (`<?>`)	762
type erasure	764

CHAPTER SUMMARY

1. *Generics* give you the capability to parameterize types. You can define a class or a method with generic types, which are substituted with concrete types.

2. The key benefit of generics is to enable errors to be detected at compile time rather than at runtime.

3. A generic class or method permits you to specify allowable types of objects that the class or method can work with. If you attempt to use a class or method with an incompatible object, the compiler will detect the error.

4. A generic type defined in a class, interface, or a static method is called a *formal generic type*, which can be replaced later with an *actual concrete type*. Replacing a generic type is called a *generic instantiation*.

5. A generic class such as `ArrayList` used without a type parameter is called a *raw type*. Use of raw types allows for backward compatibility with the earlier versions of Java.

6. A wildcard generic type has three forms: `?`, `? extends T`, and `? super T`, where `T` is a generic type. The first form, `?`, called an *unbounded wildcard*, is the same as `? extends Object`. The second form, `? extends T`, called a *bounded wildcard*, represents `T` or a subtype of `T`. The third form, `? super T`, called a *lower bound wildcard*, denotes `T` or a supertype of `T`.

7. Generics are implemented using an approach called *type erasure*. The compiler uses the generic-type information to compile the code but erases it afterward, so the generic information is not available at runtime. This approach enables the generic code to be backward compatible with the legacy code that uses raw types.

8. You cannot create an instance using a generic-type parameter such as `new E()`.

9. You cannot create an array using a generic-type parameter such as `new E[10]`.

10. You cannot use a generic-type parameter of a class in a static context.

11. Generic-type parameters cannot be used in exception classes.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

19.1 (*Revising Listing19.1*) Revise the `GenericStack` class in Listing 19.1 to implement it using an array rather than an `ArrayList`. You should check the array size before adding a new element to the stack. If the array is full, create a new array that doubles the current array size and copy the elements from the current array to the new array.

19.2 (*Implement `GenericStack` using inheritance*) In Listing 19.1, `GenericStack` is implemented using composition. Define a new stack class that extends `ArrayList`.

Draw the UML diagram for the classes then implement `GenericStack`. Write a test program that prompts the user to enter five strings and displays them in reverse order.

19.3 (*Distinct elements in `ArrayList`*) Write the following method that returns a new `ArrayList`. The new list contains the nonduplicate elements from the original list.

```
public static <E> ArrayList<E> removeDuplicates(ArrayList<E> list)
```

19.4 (*Generic linear search*) Implement the following generic method for linear search:

```
public static <E extends Comparable<E>>
  int linearSearch(E[] list, E key)
```

19.5 (*Maximum element in an array*) Implement the following method that returns the maximum element in an array:

```
public static <E extends Comparable<E>> E max(E[] list)
```

Write a test program that prompts the user to enter 10 integers, invokes this method to find the max, and displays the maximum number.

19.6 (*Maximum element in a two-dimensional array*) Write a generic method that returns the maximum element in a two-dimensional array.

```
public static <E extends Comparable<E>> E max(E[][] list)
```

19.7 (*Generic binary search*) Implement the following method using binary search:

```
public static <E extends Comparable<E>>
  int binarySearch(E[] list, E key)
```

19.8 (*Shuffle* ***`ArrayList`***) Write the following method that shuffles an **`ArrayList`**:

```
public static <E> void shuffle(ArrayList<E> list)
```

19.9 (*Sort* ***`ArrayList`***) Write the following method that sorts an **`ArrayList`**:

```
public static <E extends Comparable<E>>
  void sort(ArrayList<E> list)
```

Write a test program that prompts the user to enter 10 integers, invokes this method to sort the numbers, and displays the numbers in increasing order.

19.10 (*Largest element in an* ***`ArrayList`***) Write the following method that returns the largest element in an **`ArrayList`**:

```
public static <E extends Comparable<E>> E max(ArrayList<E> list)
```

19.11 (***`ComplexMatrix`***) Use the **`Complex`** class introduced in Programming Exercise 13.17 to develop the **`ComplexMatrix`** class for performing matrix operations involving complex numbers. The **`ComplexMatrix`** class should extend the **`GenericMatrix`** class and implement the **`add`**, **`multiple`**, and **`zero`** methods. You need to modify **`GenericMatrix`** and replace every occurrence of **`Number`** by **`Object`** because **`Complex`** is not a subtype of **`Number`**. Write a test program that creates two matrices and displays the result of addition and multiplication of the matrices by invoking the **`printResult`** method.

CHAPTER

20

LISTS, STACKS, QUEUES, AND PRIORITY QUEUES

Objectives

- To explore the relationship between interfaces and classes in the Java Collections Framework hierarchy (§20.2).
- To use the common methods defined in the `Collection` interface for operating collections (§20.2).
- To use the `Iterator` interface to traverse the elements in a collection (§20.3).
- To use a foreach loop to traverse the elements in a collection (§20.3).
- To use a `forEach` method to perform an action on each element in a collection (§20.4).
- To explore how and when to use `ArrayList` or `LinkedList` to store a list of elements (§20.5).
- To compare elements using the `Comparable` interface and the `Comparator` interface (§20.6).
- To use the static utility methods in the `Collections` class for sorting, searching, shuffling lists, and finding the largest and smallest element in collections (§20.7).
- To develop a multiple bouncing balls application using `ArrayList` (§20.8).
- To distinguish between `Vector` and `ArrayList` and to use the `Stack` class for creating stacks (§20.9).
- To explore the relationships among `Collection`, `Queue`, `LinkedList`, and `PriorityQueue` and to create priority queues using the `PriorityQueue` class (§20.10).
- To use stacks to write a program to evaluate expressions (§20.11).

20.1 Introduction

Key Point

Choosing the best data structures and algorithms for a particular task is one of the keys to developing high-performance software.

data structure

why learning data structure

Chapters 18–29 are typically taught in a data structures course. A *data structure* is a collection of data organized in some fashion. The structure not only stores data, but also supports operations for accessing and manipulating the data. Without knowing data structures, you can still write programs, but your program may not be efficient. With a good knowledge of data structures, you can build efficient programs, which are important for practical applications.

container

In object-oriented thinking, a data structure, also known as a *container* or *container object*, is an object that stores other objects, referred to as data or elements. To define a data structure is essentially to define a class. The class for a data structure should use data fields to store data and provide methods to support such operations as search, insertion, and deletion. To create a data structure is therefore to create an instance from the class. You can then apply the methods on the instance to manipulate the data structure, such as inserting an element into or deleting an element from the data structure.

Java Collections Framework

Section 11.11 introduced the **`ArrayList`** class, which is a data structure to store elements in a list. Java provides several more data structures (lists, vectors, stacks, queues, priority queues, sets, and maps) that can be used to organize and manipulate data efficiently. These are commonly known as *Java Collections Framework*. We will introduce the applications of lists, vectors, stacks, queues, and priority queues in this chapter and sets and maps in the next chapter. The implementation of these data structures will be discussed in Chapters 24–27. Through implementation, students gain a deep understanding on the efficiency of data structures and on how and when to use certain data structures. Finally, we will introduce design and implement data structures and algorithms for graphs in Chapters 28 and 29.

20.2 Collections

Key Point

*The **`Collection`** interface defines the common operations for lists, vectors, stacks, queues, priority queues, and sets.*

The Java Collections Framework supports two types of containers:

collection

map

- One for storing a collection of elements is simply called a *collection*.
- The other, for storing key/value pairs, is called a *map*.

Maps are efficient data structures for quickly searching an element using a key. We will introduce maps in the next chapter. Now we turn our attention to the following collections:

`Set`

`List`

`Stack`

`Queue`

`PrioriryQueue`

- **`Set`**s store a group of nonduplicate elements.
- **`List`**s store an ordered collection of elements.
- **`Stack`**s store objects that are processed in a last-in, first-out fashion.
- **`Queue`**s store objects that are processed in a first-in, first-out fashion.
- **`PriorityQueue`**s store objects that are processed in the order of their priorities.

The common operations of these collections are defined in the interfaces, and implementations are provided in concrete classes, as shown in Figure 20.1.

Note

All the interfaces and classes defined in the Java Collections Framework are grouped in the **`java.util`** package.

Design Guide

The design of the Java Collections Framework is an excellent example of using interfaces, abstract classes, and concrete classes. The interfaces define the common operations.

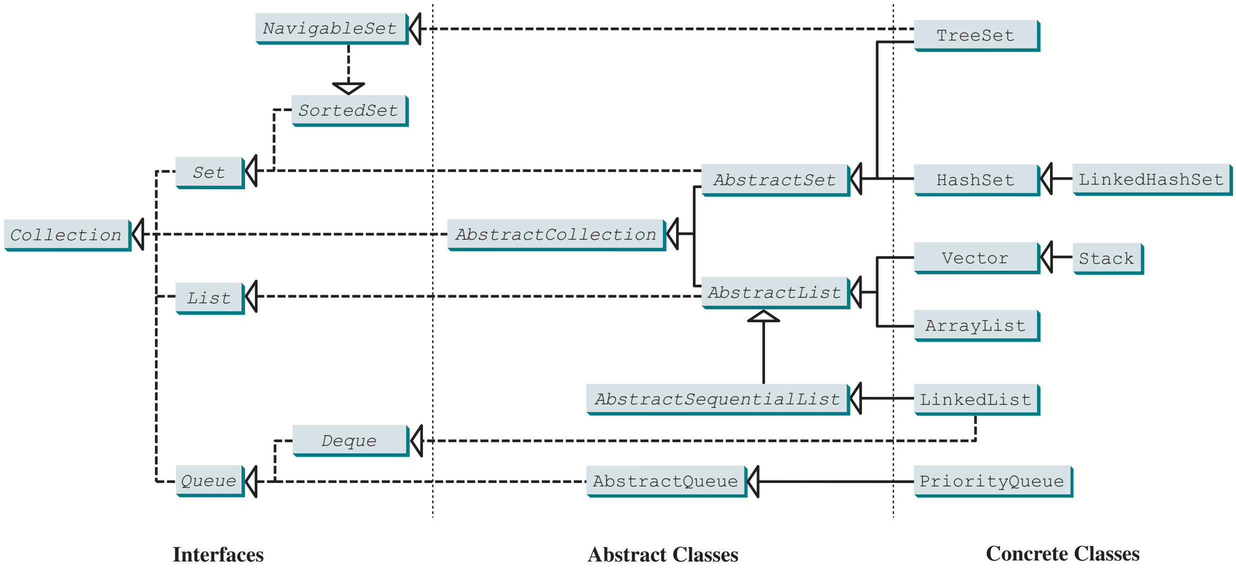

NavigableSet
TreeSet
SortedSet
Set
AbstractSet
HashSet
LinkedHashSet
Collection
AbstractCollection
Vector
Stack
List
AbstractList
ArrayList
AbstractSequentialList
LinkedList
Deque
Queue
AbstractQueue
PriorityQueue
Interfaces
Abstract Classes
Concrete Classes

FIGURE 20.1 A collection is a container that stores objects.

> The abstract classes provide partial implementation. The concrete classes implement the interfaces with concrete data structures. Providing an abstract class that partially implements an interface makes it convenient for the user to write the code. The user can simply define a concrete class that extends the abstract class rather than implementing all the methods in the interface. The abstract classes such as `AbstractCollection` are provided for convenience. For this reason, they are called *convenience abstract classes*.

convenience abstract class

The `Collection` interface is the root interface for manipulating a collection of objects. Its public methods are listed in Figure 20.2. The `AbstractCollection` class provides partial implementation for the `Collection` interface. It implements all the methods in `Collection` except the `add`, `size`, and `iterator` methods. These are implemented in the concrete subclasses.

basic operations

The `Collection` interface provides the basic operations for adding and removing elements in a collection. The `add` method adds an element to the collection. The `addAll` method adds all the elements in the specified collection to this collection. The `remove` method removes an element from the collection. The `removeAll` method removes the elements from this collection that are present in the specified collection. The `retainAll` method retains the elements in this collection that are also present in the specified collection. All these methods return `boolean`. The return value is `true` if the collection is changed as a result of the method execution. The `clear()` method simply removes all the elements from the collection.

Note

The methods `addAll`, `removeAll`, and `retainAll` are similar to the set union, difference, and intersection operations.

set operations

query operations

The `Collection` interface provides various query operations. The `size` method returns the number of elements in the collection. The `contains` method checks whether the collection contains the specified element. The `containsAll` method checks whether the collection contains all the elements in the specified collection. The `isEmpty` method returns `true` if the collection is empty.

The `Collection` interface provides the `toArray()` method, which returns an array of `Object` for the collection. It also provides the `toArray(T[])` method, which returns an array of the `T[]` type.

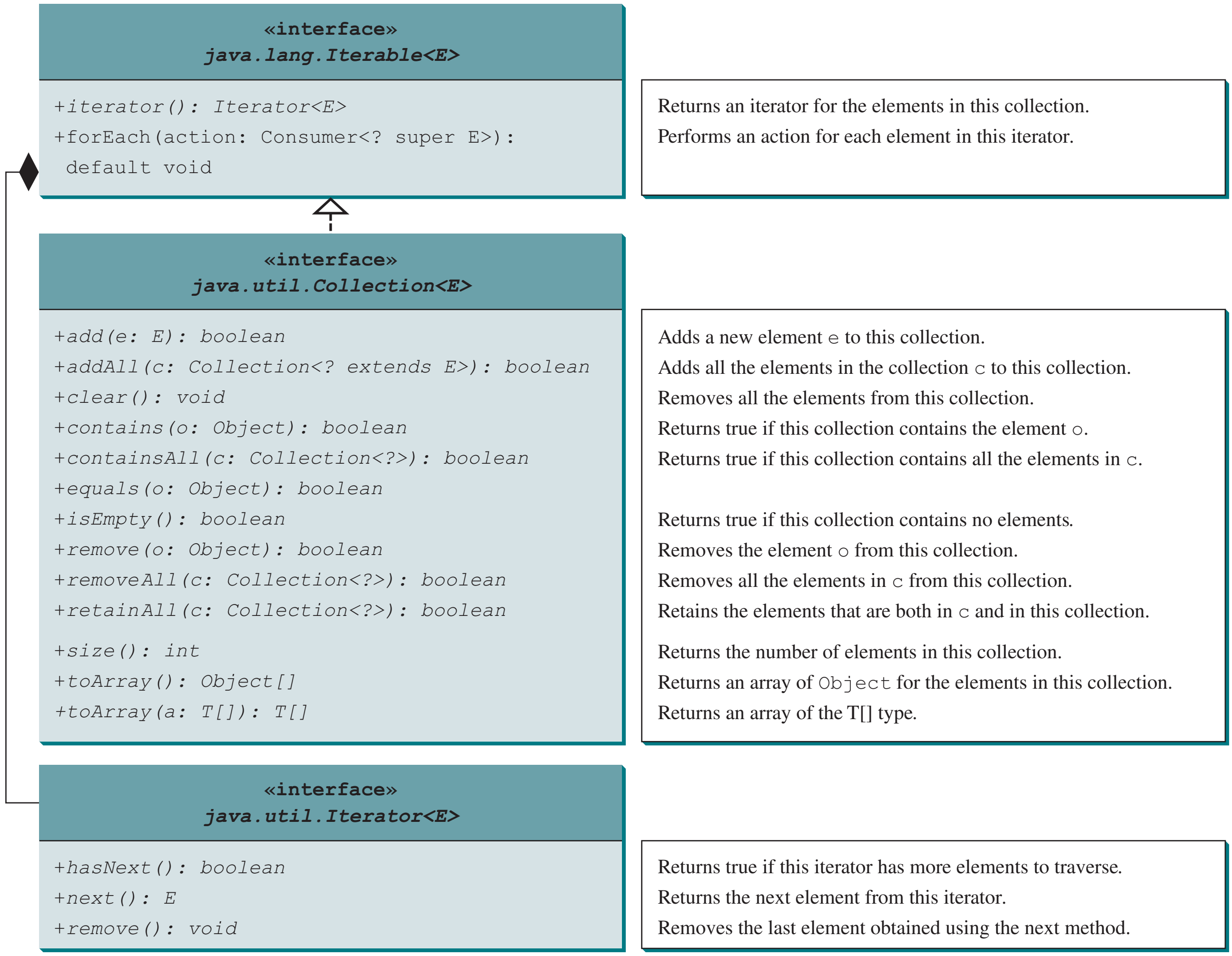

«interface»
java.lang.Iterable<E>
+iterator(): Iterator<E>
+forEach(action: Consumer<? super E>): default void
Returns an iterator for the elements in this collection.
Performs an action for each element in this iterator.
«interface»
java.util.Collection<E>
+add(e: E): boolean
+addAll(c: Collection<? extends E>): boolean
+clear(): void
+contains(o: Object): boolean
+containsAll(c: Collection<?>): boolean
+equals(o: Object): boolean
+isEmpty(): boolean
+remove(o: Object): boolean
+removeAll(c: Collection<?>): boolean
+retainAll(c: Collection<?>): boolean
+size(): int
+toArray(): Object[]
+toArray(a: T[]): T[]
Adds a new element e to this collection.
Adds all the elements in the collection c to this collection.
Removes all the elements from this collection.
Returns true if this collection contains the element o.
Returns true if this collection contains all the elements in c.
Returns true if this collection contains no elements.
Removes the element o from this collection.
Removes all the elements in c from this collection.
Retains the elements that are both in c and in this collection.
Returns the number of elements in this collection.
Returns an array of Object for the elements in this collection.
Returns an array of the T[] type.
«interface»
java.util.Iterator<E>
+hasNext(): boolean
+next(): E
+remove(): void
Returns true if this iterator has more elements to traverse.
Returns the next element from this iterator.
Removes the last element obtained using the next method.

FIGURE 20.2 The `Collection` interface contains the methods for manipulating the elements in a collection, and you can obtain an iterator object for traversing elements in the collection.

unsupported operations

Design Guide

Some of the methods in the `Collection` interface cannot be implemented in the concrete subclass. In this case, the method would throw `java.lang.UnsupportedOperationException`, a subclass of `RuntimeException`. This is a good design you can use in your project. If a method has no meaning in the subclass, you can implement it as follows:

```
public void someMethod() {
  throw new UnsupportedOperationException
    ("Method not supported");
}
```

Listing 20.1 gives an example to use the methods defined in the `Collection` interface.

LISTING 20.1 TestCollection.java

create an array list

```
import java.util.*;

public class TestCollection {
  public static void main(String[] args) {
    ArrayList<String> collection1 = new ArrayList<>();
```

```
 6         collection1.add("New York");
 7         collection1.add("Atlanta");
 8         collection1.add("Dallas");
 9         collection1.add("Madison");
10
11         System.out.println("A list of cities in collection1:");
12         System.out.println(collection1);
13
14         System.out.println("\nIs Dallas in collection1? "
15           + collection1.contains("Dallas"));
16
17         collection1.remove("Dallas");
18         System.out.println("\n" + collection1.size() +
19            " cities are in collection1 now");
20
21         Collection<String> collection2 = new ArrayList<>();
22         collection2.add("Seattle");
23         collection2.add("Portland");
24         collection2.add("Los Angeles");
25         collection2.add("Atlanta");
26
27         System.out.println("\nA list of cities in collection2:");
28         System.out.println(collection2);
29
30         ArrayList<String> c1 = (ArrayList<String>)(collection1.clone());
31         c1.addAll(collection2);
32         System.out.println("\nCities in collection1 or collection2: ");
33         System.out.println(c1);
34
35         c1 = (ArrayList<String>)(collection1.clone());
36         c1.retainAll(collection2);
37         System.out.print("\nCities in collection1 and collection2: ");
38         System.out.println(c1);
39
40         c1 = (ArrayList<String>)(collection1.clone());
41         c1.removeAll(collection2);
42         System.out.print("\nCities in collection1, but not in 2: ");
43         System.out.println(c1);
44       }
45     }
```

add elements

contains?

size?

clone

`addAll`

`retainAll`

`removeAll`

```
A list of cities in collection1:
[New York, Atlanta, Dallas, Madison]
Is Dallas in collection1? true
3 cities are in collection1 now
A list of cities in collection2:
[Seattle, Portland, Los Angeles, Atlanta]
Cities in collection1 or collection2:
[New York, Atlanta, Madison, Seattle, Portland, Los Angeles, Atlanta]
Cities in collection1 and collection2: [Atlanta]
Cities in collection1, but not in 2: [New York, Madison]
```

The program creates a concrete collection object using `ArrayList` (line 5) and invokes the `Collection` interface's `contains` method (line 15), `remove` method (line 17), `size` method (line 18), `addAll` method (line 31), `retainAll` method (line 36), and `removeAll` method (line 41).

For this example, we use `ArrayList`. You can use any concrete class of `Collection` such as `HashSet` and `LinkedList` to replace `ArrayList` to test these methods defined in the `Collection` interface.

The program creates a copy of an array list (lines 30, 35, and 40). The purpose of this is to keep the original array list intact and use its copy to perform `addAll`, `retainAll`, and `removeAll` operations.

`Cloneable`
`Serializable`

Note

All the concrete classes in the Java Collections Framework implement the `java.lang.Cloneable` and `java.io.Serializable` interfaces except that `java.util.PriorityQueue` does not implement the `Cloneable` interface. Thus, all instances of `Collection` except priority queues can be cloned and all instances of `Collection` can be serialized.

Check Point

20.2.1 What is a data structure?

20.2.2 Describe the Java Collections Framework. List the interfaces, convenience abstract classes, and concrete classes under the `Collection` interface.

20.2.3 Can a collection object be cloned and serialized?

20.2.4 What method do you use to add all the elements from one collection to another collection?

20.2.5 When should a method throw an `UnsupportedOperationException`?

20.3 Iterators

Key Point

Each collection is `Iterable`. You can obtain its `Iterator` object to traverse all the elements in the collection.

`Iterator` is a classic design pattern for walking through a data structure without having to expose the details of how data is stored in the data structure.

The `Collection` interface extends the `Iterable` interface. The `Iterable` interface defines the `iterator` method, which returns an iterator. The `Iterator` interface provides a uniform way for traversing elements in various types of collections. The `iterator()` method in the `Iterable` interface returns an instance of `Iterator`, as shown in Figure 20.2, which provides sequential access to the elements in the collection using the `next()` method. You can also use the `hasNext()` method to check whether there are more elements in the iterator, and the `remove()` method to remove the last element returned by the iterator.

Listing 20.2 gives an example that uses the iterator to traverse all the elements in an array list.

LISTING 20.2 TestIterator.java

create an array list
add elements
iterator
`hasNext()`
`next()`

```java
import java.util.*;

public class TestIterator {
  public static void main(String[] args) {
    Collection<String> collection = new ArrayList<>();
    collection.add("New York");
    collection.add("Atlanta");
    collection.add("Dallas");
    collection.add("Madison");

    Iterator<String> iterator = collection.iterator();
    while (iterator.hasNext()) {
      System.out.print(iterator.next().toUpperCase() + " ");
    }
    System.out.println();
  }
}
```

```
NEW YORK ATLANTA DALLAS MADISON
```

The program creates a concrete collection object using `ArrayList` (line 5) and adds four strings into the list (lines 6–9). The program then obtains an iterator for the collection (line 11) and uses the iterator to traverse all the strings in the list and displays the strings in uppercase (lines 12–14).

Tip

You can simplify the code in lines 11–14 using a foreach loop without using an iterator, as follows:

foreach loop

```
for (String element: collection)
  System.out.print(element.toUpperCase() + " ");
```

This loop is read as "for each element in the collection, do the following." The foreach loop can be used for arrays (see Section 7.2.7) as well as any instance of `Iterable`.

In Java 10, you can declare a variable using `var`. The compiler automatically figures out the type of the variable based on the context. For example,

```
var x = 3;
var y = x;
```

The compiler automatically determines that variable `x` is of the type `int` from the integer value `3` assigned to `x` and determines that variable `y` is of the type `int` since `int` variable `x` is assigned to `y`. This is not a good example of using the `var` type. The real good use of the `var` type is to replace long types. For example,

```
Iterator<String> iterator = new ArrayList<String>().iterator();
```

can be replaced by

```
var iterator = new ArrayList<String>().iterator();
```

This is simple and convenient and can help avoid mistakes, because sometimes the programmer may forget what the correct type for iterator.

Caution

We purposely delay the introduction of `var` until now to avoid this useful feature to be improperly used. Don't use it for primitive data types.

Note

`var` is introduced in Java 10. Ideally, it should be a keyword. However, for backward compatibility, `var` is not a keyword and not a reserved word, but rather an identifier with special meaning as the type of a local variable declaration.

Check Point

20.3.1 How do you obtain an iterator from a collection object?

20.3.2 What method do you use to obtain an element in the collection from an iterator?

20.3.3 Can you use a foreach loop to traverse the elements in any instance of `Collection`?

20.3.4 When using a foreach loop to traverse all elements in a collection, do you need to use the `next()` or `hasNext()` methods in an iterator?

20.4 Using the `forEach` Method

Key Point

You can use the `forEach` *method to perform an action for each element in a collection.*

Java 8 added a new default method `forEach` in the `Iterable` interface. The method takes an argument for specifying the action, which is an instance of a functional interface `Consumer<? super E>`. The `Consumer` interface defines the `accept(E e)` method for performing an action on the element `e`. You can rewrite the preceding example using a `forEach` method in Listing 20.3.

Listing 20.3 `TestForEach.java`

create an array list
add elements
`forEach` method

```
 1  import java.util.*;
 2
 3  public class TestForEach {
 4    public static void main(String[] args) {
 5      Collection<String> collection = new ArrayList<>();
 6      collection.add("New York");
 7      collection.add("Atlanta");
 8      collection.add("Dallas");
 9      collection.add("Madison");
10
11      collection.forEach(e -> System.out.print(e.toUpperCase() + " "));
12    }
13  }
```

```
NEW YORK ATLANTA DALLAS MADISON
```

The statement in line 11 uses a lambda expression in (a), which is equivalent to using an anonymous inner class as shown in (b). Using a lambda expression not only simplifies the syntax but also simplifies the semantics.

```
forEach(e ->
  System.out.print(e.toUppserCase() + " "))
```

(a) Use a lambda expression

```
forEach(
  new java.util.function.Consumer<String>() {
    public void accept(String e) {
      System.out.print(e.toUpperCase() + " ");
    }
  }
)
```

(b) Use an anonymous inner class

You can write the code using a foreach loop or using a `forEach` method. Using a `forEach` is simpler in most cases.

Check Point

20.4.1 Can you use the `forEach` method on any instance of `Collection`? Where is the `forEach` method defined?

20.4.2 Suppose each element in `list` is a `StringBuilder`, write a statement using a `forEach` method to change the first character to uppercase for each element in `list`.

20.5 Lists

The `List` *interface extends the* `Collection` *interface and defines a collection for storing elements in a sequential order. To create a list, use one of its two concrete classes:* `ArrayList` *or* `LinkedList`*.*

Key Point

We used `ArrayList` to test the methods in the `Collection` interface in the preceding sections. Now, we will examine `ArrayList` in more depth. We will also introduce another useful list, `LinkedList`, in this section.

20.5.1 The Common Methods in the `List` Interface

`ArrayList` and `LinkedList` are defined under the `List` interface. The `List` interface extends `Collection` to define an ordered collection with duplicates allowed. The `List` interface adds position-oriented operations as well as a new list iterator that enables a list to be traversed bidirectionally. The methods introduced in the `List` interface are shown in Figure 20.3.

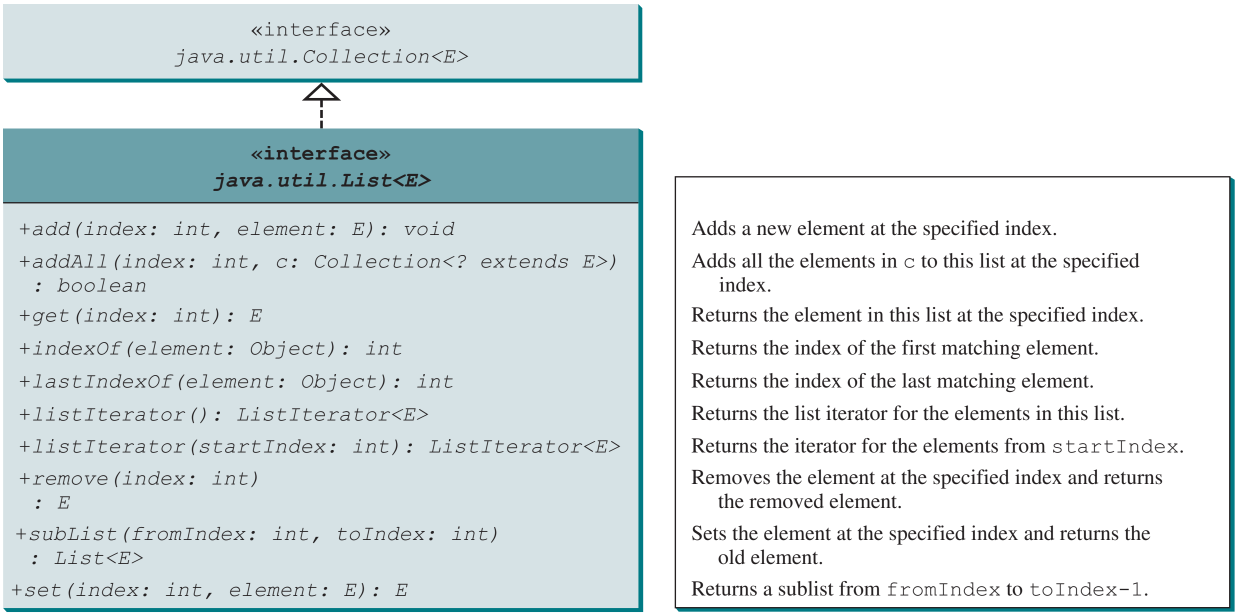

«interface»
java.util.Collection<E>
«interface»
java.util.List<E>
+add(index: int, element: E): void
+addAll(index: int, c: Collection<? extends E>) : boolean
+get(index: int): E
+indexOf(element: Object): int
+lastIndexOf(element: Object): int
+listIterator(): ListIterator<E>
+listIterator(startIndex: int): ListIterator<E>
+remove(index: int) : E
+subList(fromIndex: int, toIndex: int) : List<E>
+set(index: int, element: E): E
Adds a new element at the specified index.
Adds all the elements in c to this list at the specified index.
Returns the element in this list at the specified index.
Returns the index of the first matching element.
Returns the index of the last matching element.
Returns the list iterator for the elements in this list.
Returns the iterator for the elements from startIndex.
Removes the element at the specified index and returns the removed element.
Sets the element at the specified index and returns the old element.
Returns a sublist from fromIndex to toIndex-1.

FIGURE 20.3 The `List` interface stores elements in sequence and permits duplicates.

The `add(index, element)` method is used to insert an element at a specified index and the `addAll(index, collection)` method to insert a collection of elements at a specified index. The `remove(index)` method is used to remove an element at the specified index from the list. A new element can be set at the specified index using the `set(index, element)` method.

The `indexOf(element)` method is used to obtain the index of the specified element's first occurrence in the list and the `lastIndexOf(element)` method to obtain the index of its last occurrence. A sublist can be obtained by using the `subList(fromIndex, toIndex)` method.

The `listIterator()` or `listIterator(startIndex)` method returns an instance of `ListIterator`. The `ListIterator` interface extends the `Iterator` interface to add bidirectional traversal of the list. The methods in `ListIterator` are listed in Figure 20.4.

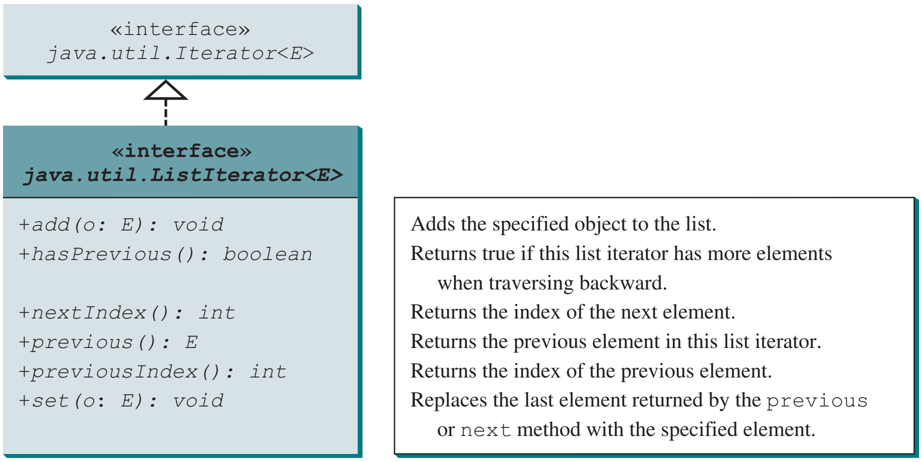

«interface»
java.util.Iterator<E>
«interface»
java.util.ListIterator<E>
+add(o: E): void
+hasPrevious(): boolean
+nextIndex(): int
+previous(): E
+previousIndex(): int
+set(o: E): void
Adds the specified object to the list.
Returns true if this list iterator has more elements when traversing backward.
Returns the index of the next element.
Returns the previous element in this list iterator.
Returns the index of the previous element.
Replaces the last element returned by the previous or next method with the specified element.

FIGURE 20.4 `ListIterator` enables traversal of a list bidirectionally.

The `add(element)` method inserts the specified element into the list. The element is inserted immediately before the next element that would be returned by the `next()` method defined in the `Iterator` interface, if any, and after the element that would be returned by the `previous()` method, if any. If the list doesn't contain any elements, the new element becomes the sole element in the list. The `set(element)` method can be used to replace the last element returned by the `next` method, or the `previous` method with the specified element.

The `hasNext()` method defined in the `Iterator` interface is used to check whether the iterator has more elements when traversed in the forward direction, and the `hasPrevious()` method to check whether the iterator has more elements when traversed in the backward direction.

The `next()` method defined in the `Iterator` interface returns the next element in the iterator, and the `previous()` method returns the previous element in the iterator. The `nextIndex()` method returns the index of the next element in the iterator, and the `previousIndex()` returns the index of the previous element in the iterator.

The `AbstractList` class provides a partial implementation for the `List` interface. The `AbstractSequentialList` class extends `AbstractList` to provide support for linked lists.

20.5.2 The `ArrayList` and `LinkedList` Classes

`ArrayList vs. LinkedList`

linked list

The `ArrayList` class and the `LinkedList` class are two concrete implementations of the `List` interface. `ArrayList` stores elements in an array. The array is dynamically created. If the capacity of the array is exceeded, a larger new array is created and all the elements from the current array are copied to the new array. `LinkedList` stores elements in a *linked list*. Which of the two classes you use depends on your specific needs. If you need to support random access through an index without inserting or removing elements at the beginning of the list, `ArrayList` is the most efficient. If, however, your application requires the insertion or deletion of elements at the beginning of the list, you should choose `LinkedList`. A list can grow or shrink dynamically. Once it is created, an array is fixed. If your application does not require the insertion or deletion of elements, an array is the most efficient data structure.

`trimToSize()`

`ArrayList` is a resizable-array implementation of the `List` interface. It also provides methods for manipulating the size of the array used internally to store the list, as shown in Figure 20.5. Each `ArrayList` instance has a capacity, which is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an `ArrayList`, its capacity grows automatically. An `ArrayList` does not automatically shrink. You can use the `trimToSize()` method to reduce the array capacity to the size of the list. An `ArrayList` can be constructed using its no-arg constructor, `ArrayList(Collection)`, or `ArrayList(initialCapacity)`.

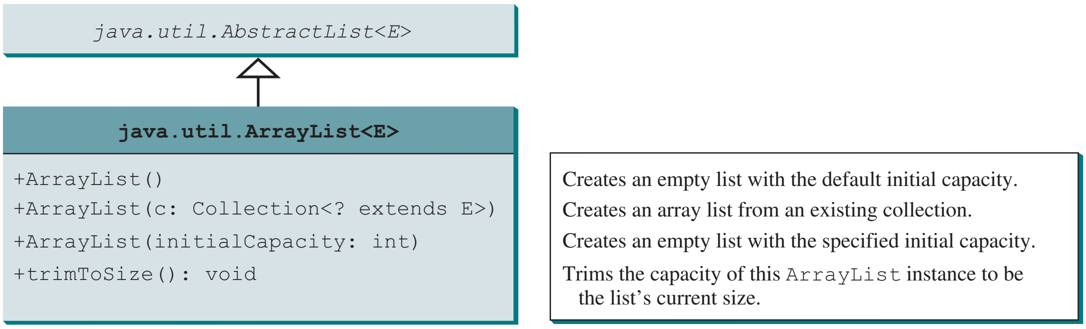

java.util.AbstractList<E>
java.util.ArrayList<E>
+ArrayList()
+ArrayList(c: Collection<? extends E>)
+ArrayList(initialCapacity: int)
+trimToSize(): void
Creates an empty list with the default initial capacity.
Creates an array list from an existing collection.
Creates an empty list with the specified initial capacity.
Trims the capacity of this ArrayList instance to be the list's current size.

FIGURE 20.5 `ArrayList` implements `List` using an array.

`LinkedList` is a linked list implementation of the `List` interface. In addition to implementing the `List` interface, this class provides the methods for retrieving, inserting, and removing elements from both ends of the list, as shown in Figure 20.6. A `LinkedList` can be constructed using its no-arg constructor or `LinkedList(Collection)`.

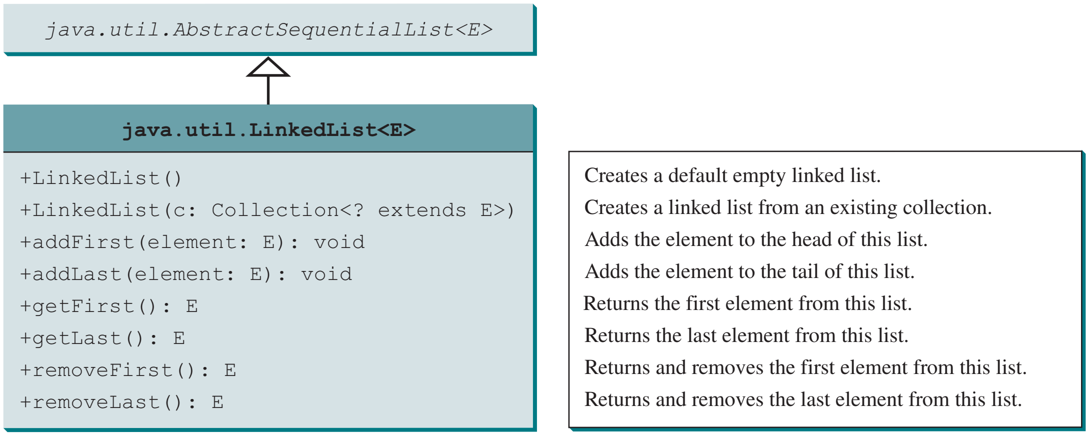

java.util.AbstractSequentialList<E>
java.util.LinkedList<E>
+LinkedList()
+LinkedList(c: Collection<? extends E>)
+addFirst(element: E): void
+addLast(element: E): void
+getFirst(): E
+getLast(): E
+removeFirst(): E
+removeLast(): E
Creates a default empty linked list.
Creates a linked list from an existing collection.
Adds the element to the head of this list.
Adds the element to the tail of this list.
Returns the first element from this list.
Returns the last element from this list.
Returns and removes the first element from this list.
Returns and removes the last element from this list.

FIGURE 20.6 `LinkedList` provides methods for adding and inserting elements at both ends of the list.

Listing 20.4 gives a program that creates an array list filled with numbers and inserts new elements into specified locations in the list. The example also creates a linked list from the array list and inserts and removes elements from the list. Finally, the example traverses the list forward and backward.

LISTING 20.4 TestArrayAndLinkedList.java

```java
import java.util.*;

public class TestArrayAndLinkedList {
  public static void main(String[] args) {
    List<Integer> arrayList = new ArrayList<>();
    arrayList.add(1); // 1 is autoboxed to an Integer object
    arrayList.add(2);
    arrayList.add(3);
    arrayList.add(1);
    arrayList.add(4);
    arrayList.add(0, 10);
    arrayList.add(3, 30);

    System.out.println("A list of integers in the array list:");
    System.out.println(arrayList);
```

array list

linked list

list iterator

list iterator

```
16
17       LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
18       linkedList.add(1, "red");
19       linkedList.removeLast();
20       linkedList.addFirst("green");
21
22       System.out.println("Display the linked list forward:");
23       ListIterator<Object> listIterator = linkedList.listIterator();
24       while (listIterator.hasNext()) {
25         System.out.print(listIterator.next() + " ");
26       }
27       System.out.println();
28
29       System.out.println("Display the linked list backward:");
30       listIterator = linkedList.listIterator(linkedList.size());
31       while (listIterator.hasPrevious()) {
32         System.out.print(listIterator.previous() + " ");
33       }
34     }
35   }
```

```
A list of integers in the array list:
[10, 1, 2, 30, 3, 1, 4]
Display the linked list forward:
green 10 red 1 2 30 3 1
Display the linked list backward:
1 3 30 2 1 red 10 green
```

A list can hold identical elements. Integer **`1`** is stored twice in the list (lines 6 and 9). **`ArrayList`** and **`LinkedList`** operate similarly. The critical difference between them pertains to internal implementation, which affects their performance. **`LinkedList`** is efficient for inserting and removing elements at the beginning of the list, and **`ArrayList`** is more efficient for all other operations. For examples of demonstrating the performance differences between **`ArrayList`** and **`LinkedList`**, see Listing.pearsoncmg.com/supplement/ArrayListvsLinkedList.pdf.

The **`get(i)`** method is available for a linked list, but it is a time-consuming operation. Do not use it to traverse all the elements in a list as shown in (a). Instead, you should use a foreach loop as shown in (b) or a **`forEach`** method as shown in (c). Note (b) and (c) use an iterator implicitly. You will know the reason when you learn how to implement a linked list in Chapter 24.

```
for (int i = 0; i < list.size(); i++)
  process list.get(i);
}
```

(a) Very inefficient

```
for (listElementType e: list) {
  process e;
}
```

(b) Efficient

```
list.forEach(e ->
  process e
)
```

(c) Efficient

`Arrays.asList(T... a)`
`method`

Tip

Java provides the static **`asList`** method for creating a list from a variable-length list of arguments. Thus, you can use the following code to create a list of strings and a list of integers:

```
List<String> list1 = Arrays.asList("red", "green", "blue");
List<Integer> list2 = Arrays.asList(10, 20, 30, 40, 50);
```

In Java 9, you can use the static **`List.of`** method to replace `Arrays.asList` for creating a list from a variable-length list of arguments. The name **`List.of`** is more intuitive and easier to memorize than **`Arrays.asList`**.

Check Point

20.5.1 How do you add and remove elements from a list? How do you traverse a list in both directions?

20.5.2 Suppose `list1` is a list that contains the strings `red`, `yellow`, and `green`, and `list2` is another list that contains the strings `red`, `yellow`, and `blue`. Answer the following questions:

a. What are `list1` and `list2` after executing `list1.addAll(list2)`?

b. What are `list1` and `list2` after executing `list1.add(list2)`?

c. What are `list1` and `list2` after executing `list1.removeAll(list2)`?

d. What are `list1` and `list2` after executing `list1.remove(list2)`?

e. What are `list1` and `list2` after executing `list1.retainAll(list2)`?

f. What is `list1` after executing `list1.clear()`?

20.5.3 What are the differences between `ArrayList` and `LinkedList`? Which list should you use to insert and delete elements at the beginning of a list?

20.5.4 Are all the methods in `ArrayList` also in `LinkedList`? What methods are in `LinkedList` but not in `ArrayList`?

20.5.5 How do you create a list from an array of objects?

20.6 The **Comparator** Interface

`Comparator` can be used to compare the objects of a class that doesn't implement `Comparable` or define a new criteria for comparing objects.

Key Point

You have learned how to compare elements using the `Comparable` interface (introduced in Section 13.6). Several classes in the Java API, such as `String`, `Date`, `Calendar`, `BigInteger`, `BigDecimal`, and all the numeric wrapper classes for the primitive types, implement the `Comparable` interface. The `Comparable` interface defines the `compareTo` method, which is used to compare two elements of the same class that implements the `Comparable` interface.

What if the elements' classes do not implement the `Comparable` interface? Can these elements be compared? You can define a *comparator* to compare the elements of different classes. To do so, define a class that implements the `java.util.Comparator<T>` interface and overrides its `compare(a, b)` method.

comparator

```
public int compare(T a, T b)
```

Returns a negative value if `a` is less than `b`, a positive value if `a` is greater than `b`, and zero if they are equal.

The `GeometricObject` class was introduced in Section 13.2, Abstract Classes. The `GeometricObject` class does not implement the `Comparable` interface. To compare the objects of the `GeometricObject` class, you can define a comparator class, as given in Listing 20.5.

Listing 20.5 GeometricObjectComparator.java

```
import java.util.Comparator;

public class GeometricObjectComparator
    implements Comparator<GeometricObject>, java.io.Serializable {
  public int compare(GeometricObject o1, GeometricObject o2) {
    double area1 = o1.getArea();
    double area2 = o2.getArea();

    if (area1 < area2)
      return -1;
```

implements `Comparator`
implements `compare`

```
11      else if (area1 == area2)
12        return 0;
13      else
14        return 1;
15    }
16  }
```

Line 4 implements `Comparator<GeometricObject>`. Line 5 overrides the `compare` method to compare two geometric objects. The class also implements `Serializable`. It is generally a good idea for comparators to implement `Serializable` so they can be serialized. The `compare(o1,o2)` method returns `1` if `o1.getArea() > o2.getArea()`, `0` if `o1.getArea() == o2.getArea()`, and `-1` otherwise.

Listing 20.6 gives a method that returns a larger object between two geometric objects. The objects are compared using the `GeometricObjectComparator`.

LISTING 20.6 TestComparator.java

```
 1  import java.util.Comparator;
 2
 3  public class TestComparator {
 4    public static void main(String[] args) {
 5      GeometricObject g1 = new Rectangle(5, 5);
 6      GeometricObject g2 = new Circle(5);
 7
 8      GeometricObject g =
 9        max(g1, g2, new GeometricObjectComparator());
10
11      System.out.println("The area of the larger object is " +
12        g.getArea());
13    }
14
15    public static GeometricObject max(GeometricObject g1,
16        GeometricObject g2, Comparator<GeometricObject> c) {
17      if (c.compare(g1, g2) > 0)
18        return g1;
19      else
20        return g2;
21    }
22  }
```

invoke `max`

the `max` method

invoke `compare`

```
The area of the larger object is 78.53981633974483
```

The program creates a `Rectangle` and a `Circle` object in lines 5 and 6 (the `Rectangle` and `Circle` classes were defined in Section 13.2, Abstract Classes). They are all subclasses of `GeometricObject`. The program invokes the `max` method to obtain the geometric object with the larger area (lines 8 and 9).

The comparator is an object of the `Comparator` type whose `compare(a,b)` method is used to compare the two elements. The `GeometricObjectComparator` is created and passed to the `max` method (line 9) and this comparator is used in the `max` method to compare the geometric objects in line 17.

Since the `Comparator` interface is a single abstract method interface, you can use a lambda expression to simplify the code by replacing line 9 with the following code

```
max(g1, g2, (o1, o2) -> o1.getArea() > o2.getArea() ?
  1 : o1.getArea() == o2.getArea() ? 0 : -1);
```

Here, `o1` and `o2` are two parameters in the `compare` method in the `Comparator` interface. The method returns `1` if `o1.getArea() > o2.getArea()`, `0` if `o1.getArea() == o2.getArea()`, and `-1` otherwise.

Note

Comparing elements using the `Comparable` interface is referred to as comparing using *natural order*, and comparing elements using the `Comparator` interface is referred to as comparing using *comparator*.

`Comparable` vs. `Comparator`
natural order
using comparator

The preceding example defines a comparator for comparing two geometric objects since the `GeometricObject` class does not implement the `Comparable` interface. Sometimes a class implements the `Comparable` interface, but if you would like to compare their objects using a different criteria, you can define a custom comparator. Listing 20.7 gives an example that compares string by their length.

Listing 20.7 `SortStringByLength.java`

```
 1  public class SortStringByLength {
 2    public static void main(String[] args) {
 3      String[] cities = {"Atlanta", "Savannah", "New York", "Dallas"};
 4      java.util.Arrays.sort(cities, new MyComparator());
 5
 6      for (String s : cities) {
 7        System.out.print(s + " ");
 8      }
 9    }
10
11    public static class MyComparator implements
12        java.util.Comparator<String> {
13      @Override
14      public int compare(String s1, String s2) {
15        return s1.length() - s2.length();
16      }
17    }
18  }
```

sort using comparator

define custom comparator

override compare method

```
Dallas Atlanta Savannah New York
```

The program defines a comparator class by implementing the `Comparator` interface (lines 11 and 12). The `compare` method is implemented to compare two strings by their lengths (lines 14–16). The program invokes the `sort` method to sort an array of strings using a comparator (line 4).

Since `Comparator` is a functional interface, the code can be simplified using a lambda expression as follows:

```
java.util.Arrays.sort(cities,
  (s1, s2) -> {return s1.length() - s2.length();});
```

or simply

```
java.util.Arrays.sort(cities,
  (s1, s2) -> s1.length() - s2.length());
```

The `List` interface defines the `sort(comparator)` method that can be used to sort the elements in a list using a specified comparator. Listing 20.8 gives an example of using a comparator to sort strings in a list by ignoring cases.

LISTING 20.8 SortStringIgnoreCase.java

lambda comparator

```
 1 public class SortStringIgnoreCase {
 2   public static void main(String[] args) {
 3     java.util.List<String> cities = java.util.Arrays.asList
 4       ("Atlanta", "Savannah", "New York", "Dallas");
 5     cities.sort((s1, s2) -> s1.compareToIgnoreCase(s2));
 6 
 7     for (String s: cities) {
 8       System.out.print(s + " ");
 9     }
10   }
11 }
```

```
Atlanta dallas new York Savannah
```

The program sorts a list of strings using a comparator that compares strings ignoring case (line 5). If you invoke `list.sort(null)`, the list will be sorted using its natural order.

The comparator is created using a lambda expression. Note the lambda expression here does nothing but simply invokes the `compareToIgnoreCase` method. In the case like this, you can use a simpler and clearer syntax to replace the lambda expression as follows:

```
cities.sort(String::compareToIgnoreCase);
```

method reference

Here `String::compareToIgnoreCase` is known as *method reference*, which is equivalent to a lambda expression. The compiler automatically translates a method reference to an equivalent lambda expression.

Comparator.comparing method

The `Comparator` interface also contains several useful static methods and default methods. You can use the static `comparing(Function<? sup T, ? sup R> keyExtracter)` method to create a `Comparator<T>` that compares the elements using the key extracted from a `Function` object. The `Function` object's `apply(T)` method returns the key of type `R` for the object `T`. For example, the following code in (a) creates a `Comparator` that compares strings by their length using a lambda expression, which is equivalent to the code using an anonymous inner class in (b) and a method reference in (c).

```
Comparator.comparing(e -> e.length())
```

(a) Use a lambda expression

```
Comparator.comparing(
  new java.util.function.Function<String, Integer>() {
    public Integer apply(String s) {
      return s.length();
    }
  })
```

(b) Use an anonymous inner class

```
Comparator.comparing(String::length)
```

(c) Use a method reference

The `comparing` method in the `Comparator` interface is implemented essentially as follows for the preceding example:

```
// comparing returns a Comparator
public static Comparator<String> comparing(Function<String, Integer> f) {
  return (s1, s2) -> f.apply(s1).compareTo(f.apply(s2));
}
```

You can replace the comparator in Listing 20.7 using the following code:

```
java.util.Arrays.sort(cities, Comparator.comparing(String::length));
```

The `Comparator.comparing` method is particularly useful to create a `Comparator` using a property from an object. For example, the following code sorts a list of `Loan` objects (see Listing 10.2) based on their `loanAmount` property.

```
Loan[] list = {new Loan(5.5, 10, 2323), new Loan(5, 10, 1000)};
Arrays.sort(list, Comparator.comparing(Loan::getLoanAmount));
```

thenComparing method

You can sort using a primary criteria, second, third, and so on using the `Comparator`'s default `thenComparing` method. For example, the following code sorts a list of `Loan` objects first on their `loanAmount` then on `annualInterestRate`.

```
Loan[] list = {new Loan(5.5, 10, 100), new Loan(5, 10, 1000)};
Arrays.sort(list, Comparator.comparing(Loan::getLoanAmount)
  .thenComparing(Loan::getAnnualInterestRate));
```

The default `reversed()` method can be used to reverse the order for a comparator. For example, the following code sorts a list of `Loan` objects on their `loanAmount` property in a decreasing order.

```
Arrays.sort(list, Comparator.comparing(Loan::getLoanAmount).
  reversed());
```

Check Point

20.6.1 What are the differences between the `Comparable` interface and the `Comparator` interface? In which package is `Comparable`, and in which package is `Comparator`?

20.6.2 How do you define a class `A` that implements the `Comparable` interface? Are two instances of class `A` comparable? How do you define a class `B` that implements the `Comparator` interface, and override the `compare` method to compare two objects of type `B1`? How do you invoke the `sort` method to sort a list of objects of the type `B1` using a comparator?

20.6.3 Write a lambda expression to create a comparator that compares two `Loan` objects by their `annualInterestRate`. Create a comparator using the `Comparator.comparing` method to compare `Loan` objects on `annualInterestRate`. Create a comparator to compare `Loan` objects first on `annualInterestRate` then on `loanAmount`.

20.6.4 Create a comparator using a lambda expression and the `Comparator.comparing` method, respectively, to compare `Collection` objects on their size.

20.6.5 Write a statement that sorts an array of `Point2D` objects on their *`y`* values and then on their *`x`* values.

20.6.6 Write a statement that sorts an `ArrayList` of strings named `list` in increasing order of their last character.

20.6.7 Write a statement that sorts a two-dimensional array of `double[][]` in increasing order of their second column. For example, if the array is `double[][] x = {{3, 1}, {2, -1}, {2, 0}}`, the sorted array will be `{{2, -1}, {2, 0}, {3, 1}}`.

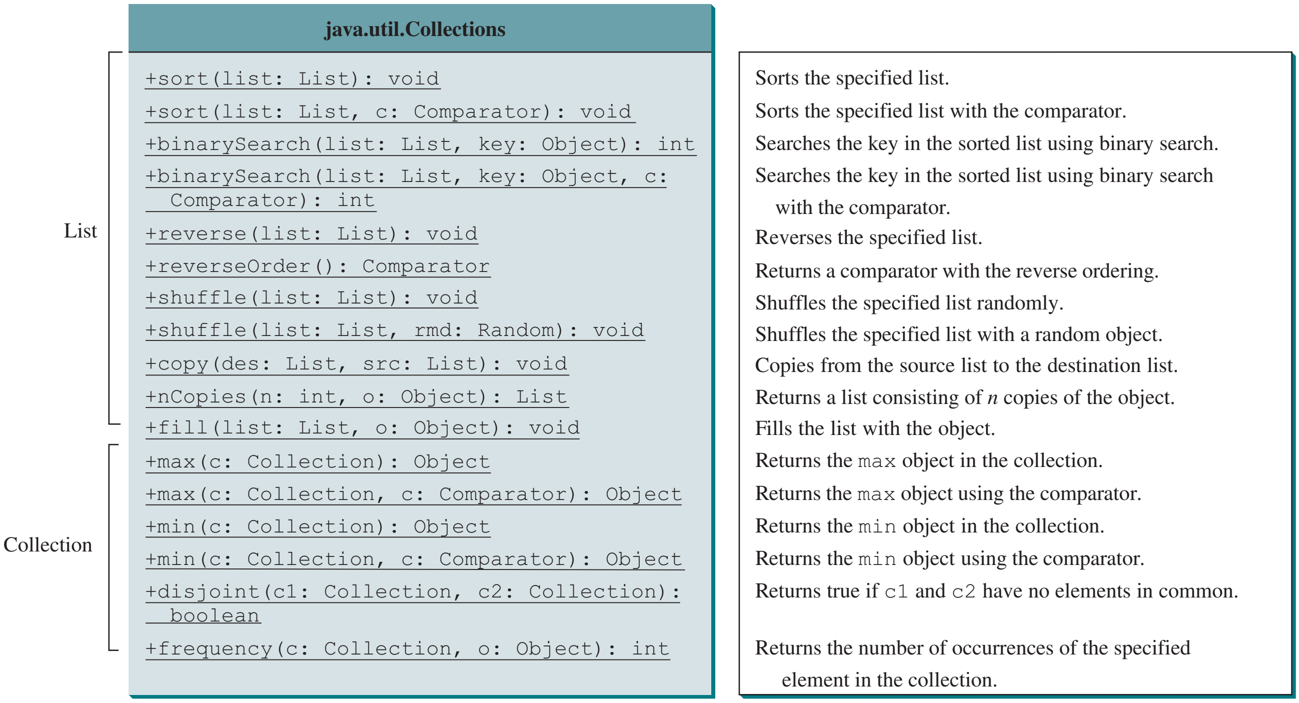
20.6.8 Write a statement that sorts a two-dimensional array of `double[][]` in increasing order of their second column as the primary order and the first column as the secondary order. For example, if the array is `double[][] x = {{3, 1}, {2, -1}, {2, 0}, {1, -1}}`, the sorted array will be `{{1, -1}, {2, -1}, {2, 0}, {3, 1}}`.

20.7 Static Methods for Lists and Collections

Key Point

The `Collections` class contains static methods to perform common operations in a collection and a list.

Section 11.12 introduced several static methods in the `Collections` class for array lists. The `Collections` class contains the `sort`, `binarySearch`, `reverse`, `shuffle`, `copy`, and `fill` methods for lists and `max`, `min`, `disjoint`, and `frequency` methods for collections, as shown in Figure 20.7.



java.util.Collections

List	`+sort(list: List): void`	Sorts the specified list.
	`+sort(list: List, c: Comparator): void`	Sorts the specified list with the comparator.
	`+binarySearch(list: List, key: Object): int`	Searches the key in the sorted list using binary search.
	`+binarySearch(list: List, key: Object, c: Comparator): int`	Searches the key in the sorted list using binary search with the comparator.
	`+reverse(list: List): void`	Reverses the specified list.
	`+reverseOrder(): Comparator`	Returns a comparator with the reverse ordering.
	`+shuffle(list: List): void`	Shuffles the specified list randomly.
	`+shuffle(list: List, rmd: Random): void`	Shuffles the specified list with a random object.
	`+copy(des: List, src: List): void`	Copies from the source list to the destination list.
	`+nCopies(n: int, o: Object): List`	Returns a list consisting of *n* copies of the object.
	`+fill(list: List, o: Object): void`	Fills the list with the object.
Collection	`+max(c: Collection): Object`	Returns the `max` object in the collection.
	`+max(c: Collection, c: Comparator): Object`	Returns the `max` object using the comparator.
	`+min(c: Collection): Object`	Returns the `min` object in the collection.
	`+min(c: Collection, c: Comparator): Object`	Returns the `min` object using the comparator.
	`+disjoint(c1: Collection, c2: Collection): boolean`	Returns true if `c1` and `c2` have no elements in common.
	`+frequency(c: Collection, o: Object): int`	Returns the number of occurrences of the specified element in the collection.

FIGURE 20.7 The `Collections` class contains static methods for manipulating lists and collections.

sort list

You can sort the comparable elements in a list in its natural order with the `compareTo` method in the `Comparable` interface. You may also specify a comparator to sort elements. For example, the following code sorts strings in a list:

```
List<String> list = Arrays.asList("red", "green", "blue");
Collections.sort(list);
System.out.println(list);
```

The output is `[blue, green, red]`.

ascending order
descending order

The preceding code sorts a list in ascending order. To sort it in descending order, you can simply use the `Collections.reverseOrder()` method to return a `Comparator` object that orders the elements in reverse of natural order. For example, the following code sorts a list of strings in descending order:

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.sort(list, Collections.reverseOrder());
System.out.println(list);
```

The output is `[yellow, red, green, blue]`.

`binarySearch`

You can use the `binarySearch` method to search for a key in a list. To use this method, the list must be sorted in increasing order. If the key is not in the list, the method returns $-(\textit{insertion point} + 1)$. Recall that the insertion point is where the item would fall in the list if it were present. For example, the following code searches the keys in a list of integers and a sorted list of strings:

```
List<Integer> list1 =
  Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
System.out.println("(1) Index: " + Collections.binarySearch(list1, 7));
System.out.println("(2) Index: " + Collections.binarySearch(list1, 9));

List<String> list2 = Arrays.asList("blue", "green", "red");
System.out.println("(3) Index: " +
  Collections.binarySearch(list2, "red"));
System.out.println("(4) Index: " +
  Collections.binarySearch(list2, "cyan"));
```

The output of the preceding code is:

```
(1) Index: 2
(2) Index: -4
(3) Index: 2
```

```
(4) Index: -2
```

reverse

You can use the **`reverse`** method to reverse the elements in a list. For example, the following code displays **`[blue, green, red, yellow]`**:

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.reverse(list);
System.out.println(list);
```

shuffle

You can use the **`shuffle(List)`** method to randomly reorder the elements in a list. For example, the following code shuffles the elements in **`list`**:

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.shuffle(list);
System.out.println(list);
```

You can also use the **`shuffle(List, Random)`** method to randomly reorder the elements in a list with a specified **`Random`** object. Using a specified **`Random`** object is useful to generate a list with identical sequences of elements for the same original list. For example, the following code shuffles the elements in **`list`**:

```
List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
List<String> list2 = Arrays.asList("yellow", "red", "green", "blue");
Collections.shuffle(list1, new Random(20));
Collections.shuffle(list2, new Random(20));
System.out.println(list1);
System.out.println(list2);
```

You will see that **`list1`** and **`list2`** have the same sequence of elements before and after the shuffling.

copy

You can use the **`copy(dest, src)`** method to copy all the elements from a source list to a destination list on the same index. The destination list must be as long as the source list. If it is longer, the remaining elements in the destination list are not affected. For example, the following code copies **`list2`** to **`list1`**:

```
List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
List<String> list2 = Arrays.asList("white", "black");
Collections.copy(list1, list2);
System.out.println(list1);
```

The output for **`list1`** is **`[white, black, green, blue]`**. The **`copy`** method performs a shallow copy: only the references of the elements from the source list are copied.

`nCopies`

You can use the **`nCopies(int n, Object o)`** method to create an immutable list that consists of **`n`** copies of the specified object. For example, the following code creates a list with five **`Calendar`** objects:

```
List<GregorianCalendar> list1 = Collections.nCopies
    (5, new GregorianCalendar(2005, 0, 1));
```

The list created from the **`nCopies`** method is immutable, so you cannot add, remove, or update elements in the list. All the elements have the same references.

`fill`

You can use the **`fill(List list, Object o)`** method to replace all the elements in the list with the specified element. For example, the following code displays **`[black, black, black]`**:

```
List<String> list = Arrays.asList("red", "green", "blue");
Collections.fill(list, "black");
System.out.println(list);
```

`max and min methods`

You can use the **`max`** and **`min`** methods for finding the maximum and minimum elements in a collection. The elements must be comparable using the **`Comparable`** interface or the **`Comparator`** interface. See the following code for examples:

```
Collection<String> collection = Arrays.asList("red", "green", "blue");
System.out.println(Collections.max(collection));     // Use Comparable
System.out.println(Collections.min(collection,
        Comparator.comparing(String::length)));  // Use Comparator
```

`disjoint method`

The **`disjoint(collection1, collection2)`** method returns **`true`** if the two collections have no elements in common. For example, in the following code, **`disjoint(collection1, collection2)`** returns **`false`**, but **`disjoint(collection1, collection3)`** returns **`true`**:

```
Collection<String> collection1 = Arrays.asList("red", "cyan");
Collection<String> collection2 = Arrays.asList("red", "blue");
Collection<String> collection3 = Arrays.asList("pink", "tan");
System.out.println(Collections.disjoint(collection1, collection2));
System.out.println(Collections.disjoint(collection1, collection3));
```

`frequency method`

The **`frequency(collection, element)`** method finds the number of occurrences of the element in the collection. For example, **`frequency(collection, "red")`** returns **`2`** in the following code:

```
Collection<String> collection = Arrays.asList("red", "cyan", "red");
System.out.println(Collections.frequency(collection, "red"));
```

Check Point

20.7.1 Are all the methods in the **`Collections`** class static?

20.7.2 Which of the following static methods in the **`Collections`** class are for lists and which are for collections?

```
sort, binarySearch, reverse, shuffle, max, min, disjoint, frequency
```

20.7.3 Show the output of the following code:

```
import java.util.*;

public class Test {
  public static void main(String[] args) {
    List<String> list =
      Arrays.asList("yellow", "red", "green", "blue");
    Collections.reverse(list);
    System.out.println(list);

    List<String> list1 =
      Arrays.asList("yellow", "red", "green", "blue");
    List<String> list2 = Arrays.asList("white", "black");
    Collections.copy(list1, list2);
    System.out.println(list1);

    Collection<String> c1 = Arrays.asList("red", "cyan");
    Collection<String> c2 = Arrays.asList("red", "blue");
    Collection<String> c3 = Arrays.asList("pink", "tan");
    System.out.println(Collections.disjoint(c1, c2));
    System.out.println(Collections.disjoint(c1, c3));

    Collection<String> collection =
      Arrays.asList("red", "cyan", "red");
    System.out.println(Collections.frequency(collection, "red"));
  }
}
```

20.7.4 Which method can you use to sort the elements in an `ArrayList` or a `LinkedList`? Which method can you use to sort an array of strings?

20.7.5 Which method can you use to perform binary search for elements in an `ArrayList` or a `LinkedList`? Which method can you use to perform binary search for an array of strings?

20.7.6 Write a statement to find the largest element in an array of comparable objects.

20.8 Case Study: Bouncing Balls

This section presents a program that displays bouncing balls and enables the user to add and remove balls.

Key Point

Section 15.12 presents a program that displays one bouncing ball. This section presents a program that displays multiple bouncing balls. You can use two buttons to suspend and resume the movement of the balls, a scroll bar to control the ball speed, and the + or − button to add or remove a ball, as shown in Figure 20.8.

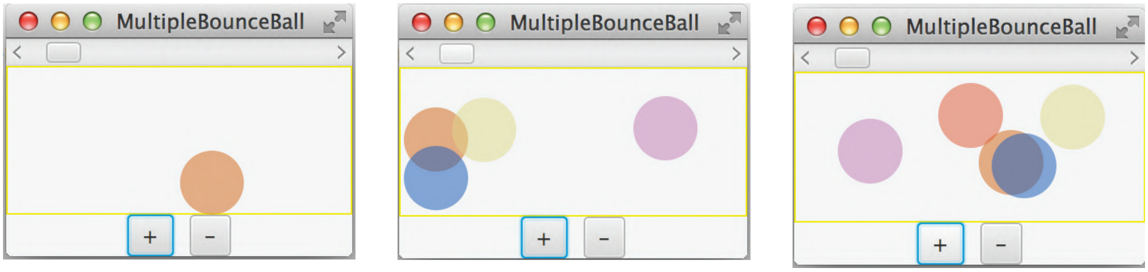


Figure 20.8 Pressing the + or − button adds or removes a ball. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

The example in Section 15.12 only had to store one ball. How do you store multiple balls in this example? The `Pane`'s `getChildren()` method returns an `ObservableList<Node>`, a subtype of `List<Node>`, for storing the nodes in the pane. Initially, the list is empty. When a new ball is created, add it to the end of the list. To remove a ball, simply remove the last one in the list.

Each ball has its state: the *x*-, *y*-coordinates, color, and direction to move. You can define a class named `Ball` that extends `javafx.scene.shape.Circle`. The *x*-, *y*-coordinates and the color are already defined in `Circle`. When a ball is created, it starts from the upper-left corner and moves downward to the right. A random color is assigned to a new ball.

The **`MultiplBallPane`** class is responsible for displaying the ball and the **`MultipleBounceBall`** class places the control components and implements the control. The relationship of these classes is shown in Figure 20.9. Listing 20.9 gives the program.

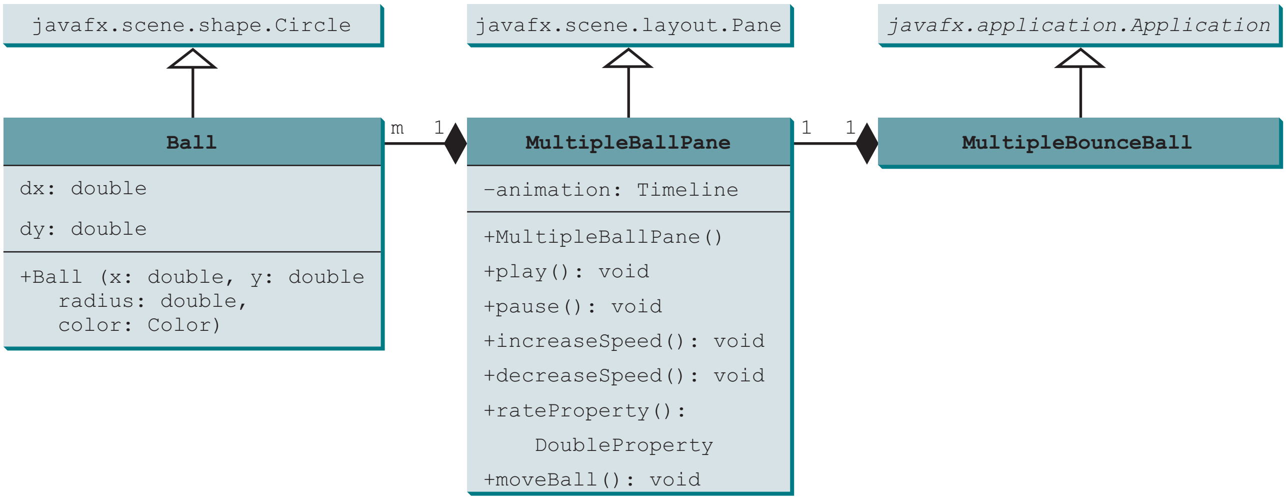

javafx.scene.shape.Circle
javafx.scene.layout.Pane
javafx.application.Application
Ball
dx: double
dy: double
+Ball (x: double, y: double radius: double, color: Color)
m
1
MultipleBallPane
-animation: Timeline
+MultipleBallPane()
+play(): void
+pause(): void
+increaseSpeed(): void
+decreaseSpeed(): void
+rateProperty(): DoubleProperty
+moveBall(): void
1
1
MultipleBounceBall

FIGURE 20.9 `MultipleBounceBall` contains `MultipleBallPane` and `MultipleBallPane` contains `Ball`.

LISTING 20.9 MultipleBounceBall.java

```
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.beans.property.DoubleProperty;
import javafx.geometry.Pos;
import javafx.scene.Node;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ScrollBar;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.util.Duration;

public class MultipleBounceBall extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    MultipleBallPane ballPane = new MultipleBallPane();
    ballPane.setStyle("-fx-border-color: yellow");

    Button btAdd = new Button("+");
    Button btSubtract = new Button("-");
    HBox hBox = new HBox(10);
    hBox.getChildren().addAll(btAdd, btSubtract);
    hBox.setAlignment(Pos.CENTER);

    // Add or remove a ball
    btAdd.setOnAction(e -> ballPane.add());
    btSubtract.setOnAction(e -> ballPane.subtract());
```

create a ball pane
set ball pane border
create buttons
add buttons to `HBox`
add a ball
remove a ball

```

    // Pause and resume animation
    ballPane.setOnMousePressed(e -> ballPane.pause());
    ballPane.setOnMouseReleased(e -> ballPane.play());

    // Use a scroll bar to control animation speed
    ScrollBar sbSpeed = new ScrollBar();
    sbSpeed.setMax(20);
    sbSpeed.setValue(10);
    ballPane.rateProperty().bind(sbSpeed.valueProperty());

    BorderPane pane = new BorderPane();
    pane.setCenter(ballPane);
    pane.setTop(sbSpeed);
    pane.setBottom(hBox);

    // Create a scene and place the pane in the stage
    Scene scene = new Scene(pane, 250, 150);
    primaryStage.setTitle("MultipleBounceBall"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  private class MultipleBallPane extends Pane {
    private Timeline animation;

    public MultipleBallPane() {
      // Create an animation for moving the ball
      animation = new Timeline(
        new KeyFrame(Duration.millis(50), e -> moveBall()));
      animation.setCycleCount(Timeline.INDEFINITE);
      animation.play(); // Start animation
    }

    public void add() {
      Color color = new Color(Math.random(),
        Math.random(), Math.random(), 0.5);
      getChildren().add(new Ball(30, 30, 20,color));
    }

    public void subtract() {
      if (getChildren().size() > 0) {
        getChildren().remove(getChildren().size() - 1);
      }
    }

    public void play() {
      animation.play();
    }

    public void pause() {
      animation.pause();
    }

    public void increaseSpeed() {
      animation.setRate(animation.getRate() + 0.1);
    }

    public void decreaseSpeed() {
      animation.setRate(
```

pause animation
resume animation
create a scroll bar
bind animation rate
add a ball to pane
remove a ball

move all balls

change *x*-direction

change *y*-direction

adjust ball positions

declare d*x* and d*y*

create a ball

```
 93       animation.getRate() > 0 ? animation.getRate() -  0.1 :  0);
 94   }
 95
 96   public DoubleProperty rateProperty() {
 97     return animation.rateProperty();
 98   }
 99
100   protected void moveBall() {
101     for (Node node:  this.getChildren()) {
102       Ball ball = (Ball)node;
103       // Check boundaries
104       if (ball.getCenterX() < ball.getRadius() ||
105           ball.getCenterX() > getWidth() - ball.getRadius()) {
106         ball.dx *= -1; // Change ball move direction
107       }
108       if (ball.getCenterY() < ball.getRadius() ||
109           ball.getCenterY() > getHeight() - ball.getRadius()) {
110         ball.dy *= -1; // Change ball move direction
111       }
112
113       // Adjust ball position
114       ball.setCenterX(ball.dx + ball.getCenterX());
115       ball.setCenterY(ball.dy + ball.getCenterY());
116     }
117   }
118  }
119
120  class Ball extends Circle {
121   private double dx = 1, dy = 1;
122
123   Ball(double x, double y, double radius, Color color) {
124     super(x, y, radius);
125     setFill(color); // Set ball color
126   }
127  }
128 }
```

The `add()` method creates a new ball with a random color and adds it to the pane (line 70). The pane stores all the balls in a list. The `subtract()` method removes the last ball in the list (line 75).

When the user clicks the + button, a new ball is added to the pane (line 31). When the user clicks the − button, the last ball in the array list is removed (line 32).

The `moveBall()` method in the `MultipleBallPane` class gets every ball in the pane's list and adjusts the balls' positions (lines 114 and 115).

Check Point

20.8.1 What is the return value from invoking `pane.getChildren()` for a pane?

20.8.2 How do you modify the code in the `MutilpleBallApp` program to remove the first ball in the list when the − button is clicked?

20.8.3 How do you modify the code in the `MutilpleBallApp` program so each ball will get a random radius between 10 and 20?

20.9 Vector and Stack Classes

Key Point

`Vector` is a subclass of `AbstractList` and `Stack` is a subclass of `Vector` in the Java API.

The Java Collections Framework was introduced in Java 2. Several data structures were supported earlier, among them the `Vector` and `Stack` classes. These classes were redesigned to fit into the Java Collections Framework, but all their old-style methods are retained for compatibility.

`Vector` is the same as `ArrayList`, except that it contains synchronized methods for accessing and modifying the vector. Synchronized methods can prevent data corruption when a vector is accessed and modified by two or more threads concurrently. We will discuss synchronization in Chapter 32, Multithreading and Parallel Programming. For the many applications that do not require synchronization, using `ArrayList` is more efficient than using `Vector`.

The `Vector` class extends the `AbstractList` class. It also has the methods contained in the original `Vector` class defined prior to Java 2, as shown in Figure 20.10.

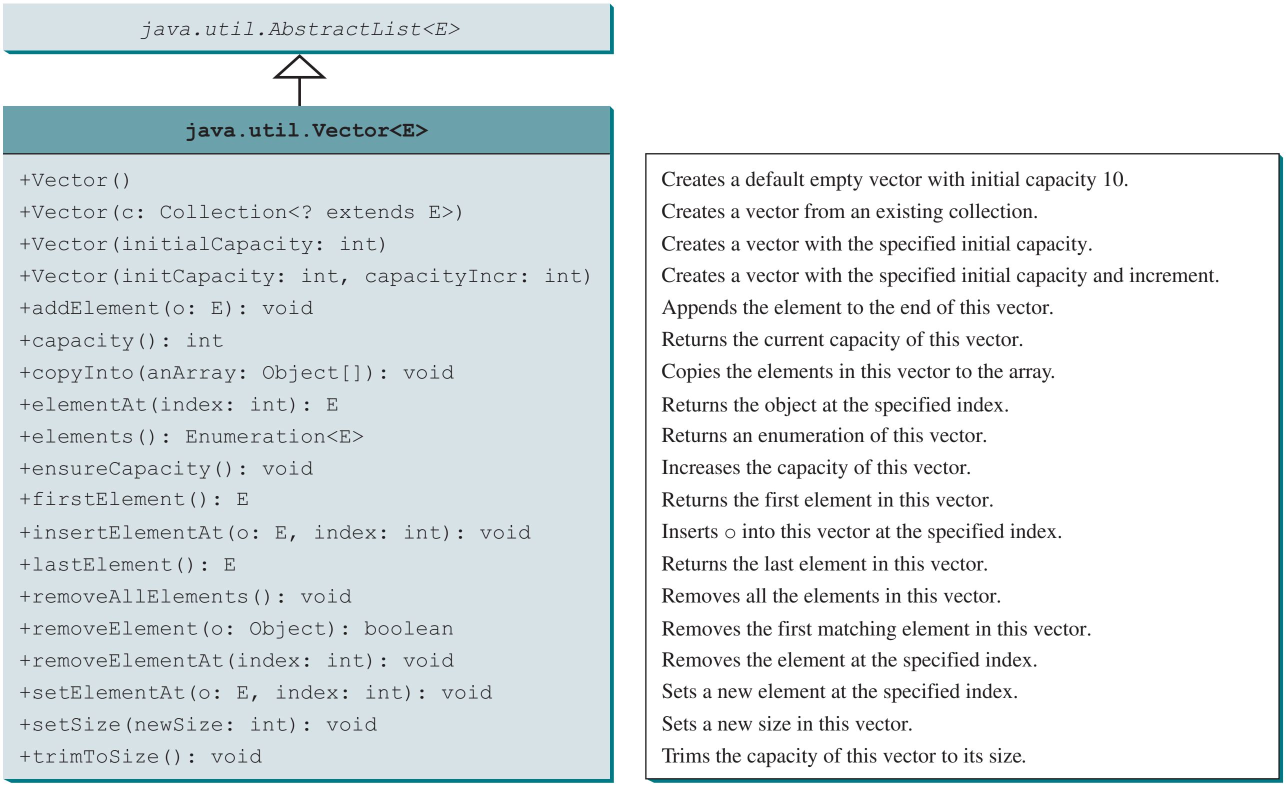

java.util.AbstractList<E>
java.util.Vector<E>
+Vector()
+Vector(c: Collection<? extends E>)
+Vector(initialCapacity: int)
+Vector(initCapacity: int, capacityIncr: int)
+addElement(o: E): void
+capacity(): int
+copyInto(anArray: Object[]): void
+elementAt(index: int): E
+elements(): Enumeration<E>
+ensureCapacity(): void
+firstElement(): E
+insertElementAt(o: E, index: int): void
+lastElement(): E
+removeAllElements(): void
+removeElement(o: Object): boolean
+removeElementAt(index: int): void
+setElementAt(o: E, index: int): void
+setSize(newSize: int): void
+trimToSize(): void
Creates a default empty vector with initial capacity 10.
Creates a vector from an existing collection.
Creates a vector with the specified initial capacity.
Creates a vector with the specified initial capacity and increment.
Appends the element to the end of this vector.
Returns the current capacity of this vector.
Copies the elements in this vector to the array.
Returns the object at the specified index.
Returns an enumeration of this vector.
Increases the capacity of this vector.
Returns the first element in this vector.
Inserts o into this vector at the specified index.
Returns the last element in this vector.
Removes all the elements in this vector.
Removes the first matching element in this vector.
Removes the element at the specified index.
Sets a new element at the specified index.
Sets a new size in this vector.
Trims the capacity of this vector to its size.

FIGURE 20.10 Starting in Java 2, the `Vector` class extends `AbstractList` and also retains all the methods in the original `Vector` class.

Most of the methods in the `Vector` class listed in the UML diagram in Figure 20.10 are similar to the methods in the `List` interface. These methods were introduced before the Java Collections Framework. For example, `addElement(Object element)` is the same as the `add(Object element)` method, except that the `addElement` method is synchronized. Use the `ArrayList` class if you don't need synchronization. It works much faster than `Vector`.

Note

The `elements()` method returns an `Enumeration`. The `Enumeration` interface was introduced prior to Java 2 and was superseded by the `Iterator` interface.

Note

`Vector` is widely used in Java legacy code because it was the Java resizable-array implementation before Java 2.

In the Java Collections Framework, `Stack` is implemented as an extension of `Vector`, as illustrated in Figure 20.11.

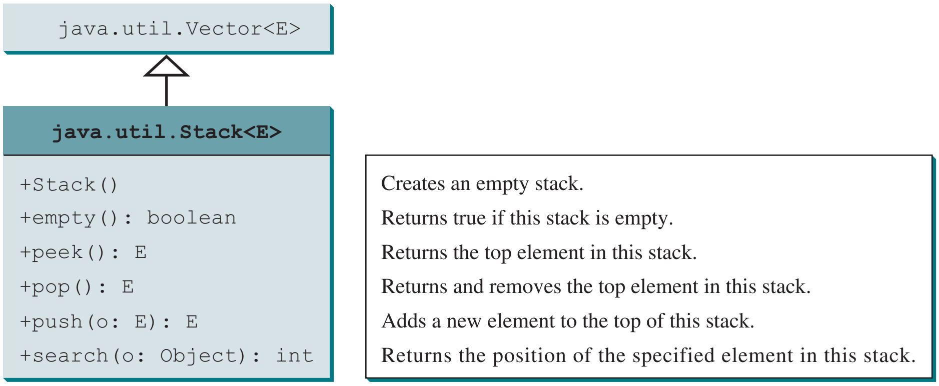

java.util.Vector<E>
java.util.Stack<E>
+Stack()
+empty(): boolean
+peek(): E
+pop(): E
+push(o: E): E
+search(o: Object): int
Creates an empty stack.
Returns true if this stack is empty.
Returns the top element in this stack.
Returns and removes the top element in this stack.
Adds a new element to the top of this stack.
Returns the position of the specified element in this stack.

FIGURE 20.11 The `Stack` class extends `Vector` to provide a last-in, first-out data structure.

The `Stack` class was introduced prior to Java 2. The methods shown in Figure 20.11 were used before Java 2. The `empty()` method is the same as `isEmpty()`. The `peek()` method looks at the element at the top of the stack without removing it. The `pop()` method removes the top element from the stack and returns it. The `push(Object element)` method adds the specified element to the stack. The `search(Object element)` method checks whether the specified element is in the stack.

Check Point

20.9.1 How do you create an instance of `Vector`? How do you add or insert a new element into a vector? How do you remove an element from a vector? How do you find the size of a vector?

20.9.2 How do you create an instance of `Stack`? How do you add a new element to a stack? How do you remove an element from a stack? How do you find the size of a stack?

20.9.3 Does Listing 20.1, TestCollection.java, compile and run if all the occurrences of `ArrayList` are replaced by `LinkedList`, `Vector`, or `Stack`?

20.10 Queues and Priority Queues

Key Point

In a priority queue, the element with the highest priority is removed first.

queue
priority queue

A *queue* is a first-in, first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a *priority queue*, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first. This section introduces queues and priority queues in the Java API.

20.10.1 The Queue Interface

Queue interface

The `Queue` interface extends `java.util.Collection` with additional insertion, extraction, and inspection operations, as shown in Figure 20.12.

queue operations

The `offer` method is used to add an element to the queue. This method is similar to the `add` method in the `Collection` interface, but the `offer` method is preferred for queues. The `poll` and `remove` methods are similar, except that `poll()` returns `null` if the queue is empty, whereas `remove()` throws an exception. The `peek` and `element` methods are similar, except that `peek()` returns `null` if the queue is empty, whereas `element()` throws an exception.

20.10.2 Deque and LinkedList

The `LinkedList` class implements the `Deque` interface, which extends the `Queue` interface, as shown in Figure 20.13. Therefore, you can use `LinkedList` to create a queue. `LinkedList` is ideal for queue operations because it is efficient for inserting and removing elements from both ends of a list.

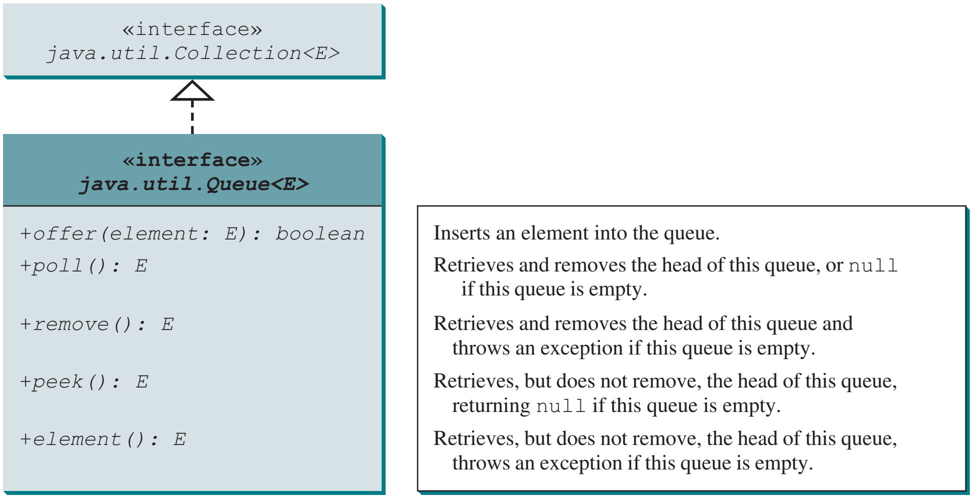

«interface»
java.util.Collection<E>
«interface»
java.util.Queue<E>
+offer(element: E): boolean
+poll(): E
+remove(): E
+peek(): E
+element(): E
Inserts an element into the queue.
Retrieves and removes the head of this queue, or null if this queue is empty.
Retrieves and removes the head of this queue and throws an exception if this queue is empty.
Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.
Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.

FIGURE 20.12 The **`Queue`** interface extends **`Collection`** to provide additional insertion, extraction, and inspection operations.

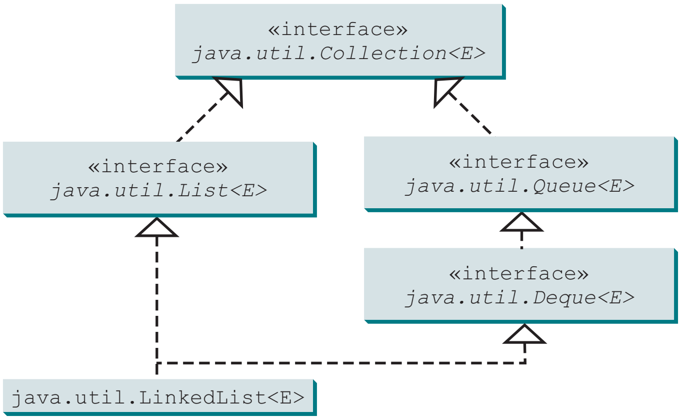

«interface»
java.util.Collection<E>
«interface»
java.util.List<E>
«interface»
java.util.Queue<E>
«interface»
java.util.Deque<E>
java.util.LinkedList<E>

FIGURE 20.13 **`LinkedList`** implements **`List`** and **`Deque`**.

`Deque` supports element insertion and removal at both ends. The name *deque* is short for "double-ended queue" and is usually pronounced "deck." The **`Deque`** interface extends **`Queue`** with additional methods for inserting and removing elements from both ends of the queue. The methods **`addFirst(e)`**, **`removeFirst()`**, **`addLast(e)`**, **`removeLast()`**, **`getFirst()`**, and **`getLast()`** are defined in the **`Deque`** interface.

Listing 20.10 shows an example of using a queue to store strings. Line 3 creates a queue using **`LinkedList`**. Four strings are added to the queue in lines 4–7. The **`size()`** method defined in the **`Collection`** interface returns the number of elements in the queue (line 9). The **`remove()`** method retrieves and removes the element at the head of the queue (line 10).

LISTING 20.10 TestQueue.java

```
1  public class TestQueue {
2    public static void main(String[] args) {
3      java.util.Queue<String> queue = new java.util.LinkedList<>();
4      queue.offer("Oklahoma");
5      queue.offer("Indiana");
6      queue.offer("Georgia");
7      queue.offer("Texas");
8
```

creates a queue
inserts an element
queue size

queue size
remove element

```
    while (queue.size() > 0)
      System.out.print(queue.remove() + " ");
  }
}
```

```
Oklahoma Indiana Georgia Texas
```

PriorityQueue class

The `PriorityQueue` class implements a priority queue, as shown in Figure 20.14. By default, the priority queue orders its elements according to their natural ordering using `Comparable`. The element with the least value is assigned the highest priority, and thus is removed from the queue first. If there are several elements with the same highest priority, the tie is broken arbitrarily. You can also specify an ordering using `Comparator` in the constructor `PriorityQueue(initialCapacity, comparator)`.

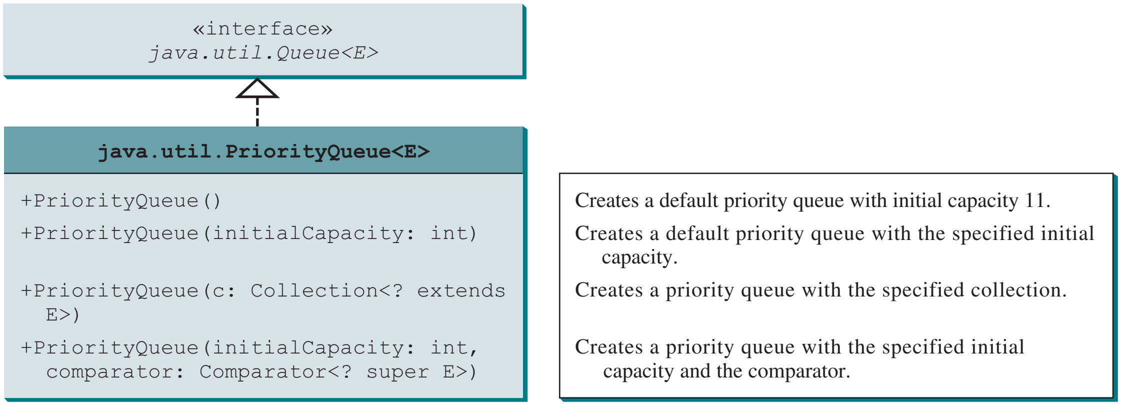

«interface»
java.util.Queue<E>
java.util.PriorityQueue<E>
+PriorityQueue()
+PriorityQueue(initialCapacity: int)
+PriorityQueue(c: Collection<? extends E>)
+PriorityQueue(initialCapacity: int, comparator: Comparator<? super E>)
Creates a default priority queue with initial capacity 11.
Creates a default priority queue with the specified initial capacity.
Creates a priority queue with the specified collection.
Creates a priority queue with the specified initial capacity and the comparator.

FIGURE 20.14 The `PriorityQueue` class implements a priority queue.

Listing 20.11 shows an example of using a priority queue to store strings. Line 5 creates a priority queue for strings using its no-arg constructor. This priority queue orders the strings using their natural order, so the strings are removed from the queue in increasing order. Lines 16 and 17 create a priority queue using the comparator obtained from `Collections.reverseOrder()`, which orders the elements in reverse order, so the strings are removed from the queue in decreasing order.

LISTING 20.11 PriorityQueueDemo.java

a default queue
inserts an element

```
 1 import java.util.*;
 2 
 3 public class PriorityQueueDemo {
 4   public static void main(String[] args) {
 5     PriorityQueue<String> queue1 = new PriorityQueue<>();
 6     queue1.offer("Oklahoma");
 7     queue1.offer("Indiana");
 8     queue1.offer("Georgia");
 9     queue1.offer("Texas");
10 
11     System.out.println("Priority queue using Comparable:");
12     while (queue1.size() > 0) {
13       System.out.print(queue1.remove() + " ");
14     }
15 
16     PriorityQueue<String> queue2 = new PriorityQueue<>(
17       4, Collections.reverseOrder());
```

```
    queue2.offer("Oklahoma");
    queue2.offer("Indiana");
    queue2.offer("Georgia");
    queue2.offer("Texas");

    System.out.println("\nPriority queue using Comparator:");
    while (queue2.size() > 0) {
      System.out.print(queue2.remove() + " ");
    }
  }
}
```

a queue with comparator

```
Priority queue using Comparable:
Georgia Indiana Oklahoma Texas
Priority queue using Comparator:
Texas Oklahoma Indiana Georgia
```

20.10.1 Is `java.util.Queue` a subinterface of `java.util.Collection`, `java.util.Set`, or `java.util.List`? Does `LinkedList` implement `Queue`?

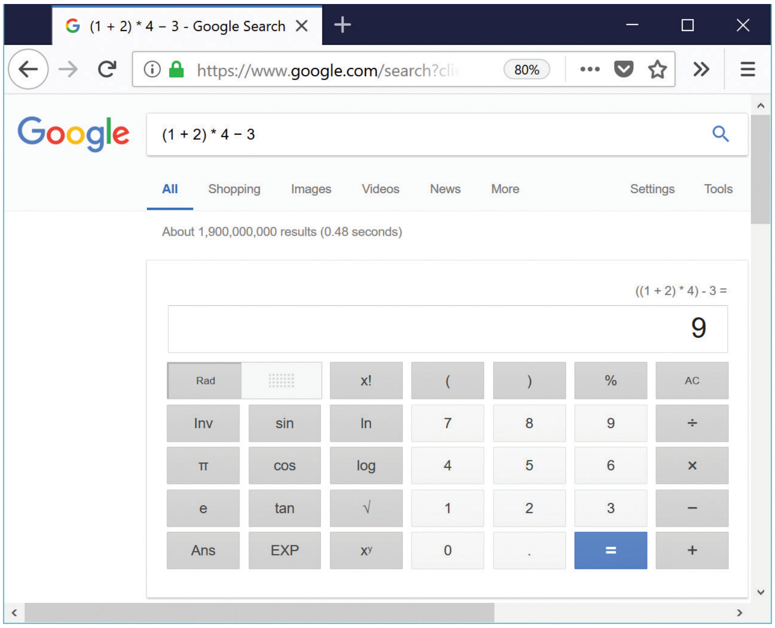
20.10.2 How do you create a priority queue for integers? By default, how are elements ordered in a priority queue? Is the element with the least value assigned the highest priority in a priority queue?

20.10.3 How do you create a priority queue that reverses the natural order of the elements?

20.11 Case Study: Evaluating Expressions

Stacks can be used to evaluate expressions.

Stacks and queues have many applications. This section gives an application that uses stacks to evaluate expressions. You can enter an arithmetic expression from Google to evaluate the expression, as shown in Figure 20.15.



(1 + 2) * 4 – 3 - Google Search

https://www.google.com/search?cli

(1 + 2) * 4 – 3

About 1,900,000,000 results (0.48 seconds)

((1 + 2) * 4) - 3 =

9

FIGURE 20.15 You can evaluate an arithmetic expression using a Google search engine.
Source: Google and the Google logo are registered trademarks of Google Inc., used with permission.

compound expression

How does Google evaluate an expression? This section presents a program that evaluates a *compound expression* with multiple operators and parentheses (e.g., `(1 + 2) * 4 - 3`). For simplicity, assume the operands are integers, and the operators are of four types: `+`, `-`, `*`, and `/`.

process an operator

The problem can be solved using two stacks, named `operandStack` and `operatorStack`, for storing operands and operators, respectively. Operands and operators are pushed into the stacks before they are processed. When an *operator is processed*, it is popped from `operatorStack` and applied to the first two operands from `operandStack` (the two operands are popped from `operandStack`). The resultant value is pushed back to `operandStack`.

The algorithm proceeds in two phases:

Phase 1: Scanning the expression

The program scans the expression from left to right to extract operands, operators, and the parentheses.

1.1. If the extracted item is an operand, push it to `operandStack`.

1.2. If the extracted item is a `+` or `-` operator, process all the operators at the top of `operatorStack` and push the extracted operator to `operatorStack`.

1.3. If the extracted item is a `*` or `/` operator, process the `*` or `/` operators at the top of `operatorStack` and push the extracted operator to `operatorStack`.

1.4. If the extracted item is a `(` symbol, push it to `operatorStack`.

1.5. If the extracted item is a `)` symbol, repeatedly process the operators from the top of `operatorStack` until seeing the `(` symbol on the stack.

Phase 2: Clearing the stack

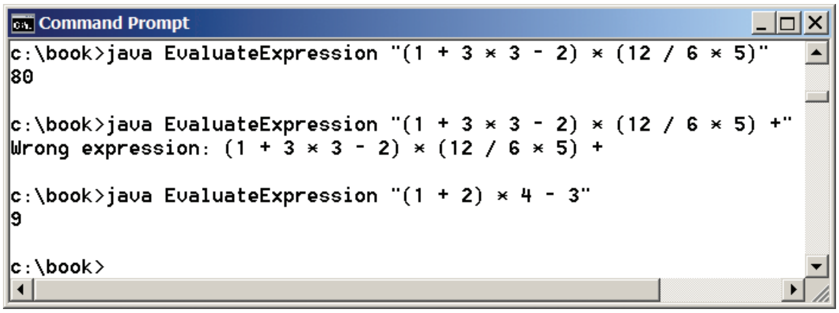
Repeatedly process the operators from the top of `operatorStack` until `operatorStack` is empty.

Table 20.1 shows how the algorithm is applied to evaluate the expression `(1 + 2) * 4 - 3`.

TABLE 20.1 Evaluating an Expression

Expression	*Scan*	*Action*	`operandStack`	`operatorStack`
(1 + 2) * 4 − 3 ↑	(	Phase 1.4		(
(1 + 2) * 4 − 3 ↑	1	Phase 1.1	1	(
(1 + 2) * 4 − 3 ↑	+	Phase 1.2	1	+ (
(1 + 2) * 4 − 3 ↑	2	Phase 1.1	2 1	(
(1 + 2) * 4 − 3 ↑	)	Phase 1.5	3	
(1 + 2) * 4 − 3 ↑	*	Phase 1.3	3	*
(1 + 2) * 4 − 3 ↑	4	Phase 1.1	4 3	*
(1 + 2) * 4 − 3 ↑	−	Phase 1.2	12	-
(1 + 2) * 4 − 3 ↑	3	Phase 1.1	3 12	-
(1 + 2) * 4 − 3 ↑	none	Phase 2	9	

Listing 20.12 gives the program, and Figure 20.16 shows some sample output.



```
c:\book>java EvaluateExpression "(1 + 3 * 3 - 2) * (12 / 6 * 5)"
80

c:\book>java EvaluateExpression "(1 + 3 * 3 - 2) * (12 / 6 * 5) +"
Wrong expression: (1 + 3 * 3 - 2) * (12 / 6 * 5) +

c:\book>java EvaluateExpression "(1 + 2) * 4 - 3"
9

c:\book>
```

Figure 20.16 The program takes an expression as command-line arguments. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Listing 20.12 `EvaluateExpression.java`

```java
import java.util.Stack;

public class EvaluateExpression {
  public static void main(String[] args) {
    // Check number of arguments passed
    if (args.length != 1) {
      System.out.println(
        "Usage: java EvaluateExpression \"expression\"");
      System.exit(1);
    }

    try {
      System.out.println(evaluateExpression(args[0]));
    }
    catch (Exception ex) {
      System.out.println("Wrong expression: " + args[0]);
    }
  }

  /** Evaluate an expression */
  public static int evaluateExpression(String expression) {
    // Create operandStack to store operands
    Stack<Integer> operandStack = new Stack<>();

    // Create operatorStack to store operators
    Stack<Character> operatorStack = new Stack<>();

    // Insert blanks around (, ), +, -, /, and *
    expression = insertBlanks(expression);

    // Extract operands and operators
    String[] tokens = expression.split(" ");

    // Phase 1: Scan tokens
    for (String token: tokens) {
      if (token.length() == 0) // Blank space
        continue; // Back to the while loop to extract the next token
      else if (token.charAt(0) == '+' || token.charAt(0) == '-') {
        // Process all +, -, *, / in the top of the operator stack
        while (!operatorStack.isEmpty() &&
          (operatorStack.peek() == '+' ||
```

check usage

evaluate expression

exception

`operandStack`

`operatorStack`

prepare for extraction

extract tokens

process tokens

+ or − scanned

```
           operatorStack.peek() == '-' ||
           operatorStack.peek() == '*' ||
           operatorStack.peek() == '/')) {
           processAnOperator(operandStack, operatorStack);
        }

        // Push the + or - operator into the operator stack
        operatorStack.push(token.charAt(0));
      }
```

* or / scanned

```
      else if (token.charAt(0) == '*' || token.charAt(0) == '/') {
        // Process all *, / in the top of the operator stack
        while (!operatorStack.isEmpty() &&
          (operatorStack.peek() == '*' ||
          operatorStack.peek() == '/')) {
          processAnOperator(operandStack, operatorStack);
        }

        // Push the * or / operator into the operator stack
        operatorStack.push(token.charAt(0));
       }
```

(scanned

```
       else if(token.trim().charAt(0) =='(') {
         operatorStack.push('('); // Push '(' to stack
      }
```

) scanned

```
      else if (token.trim().charAt(0) ==')') {
        // Process all the operators in the stack until seeing '('
        while (operatorStack.peek() != '(') {
          processAnOperator(operandStack, operatorStack);
        }
```

an operand scanned

```
        operatorStack.pop(); // Pop the '(' symbol from the stack
      }
      else {  // An operand scanned
        // Push an operand to the stack
        operandStack.push(Integer.valueOf(token));
      }
    }

    // Phase 2: Process all the remaining operators in the stack
```

clear operatorStack

```
    while (!operatorStack.isEmpty()) {
      processAnOperator(operandStack, operatorStack);
    }

    // Return the result
```

return result

```
    return operandStack.pop();
  }

  /** Process one operator: Take an operator from operatorStack and
   *  apply it on the operands in the operandStack */
  public static void processAnOperator(
      Stack<Integer> operandStack, Stack<Character> operatorStack) {
    char op = operatorStack.pop();
    int op1 = operandStack.pop();
    int op2 = operandStack.pop();
```

process +

```
    if (op == '+')
      operandStack.push(op2 + op1);
```

process −

```
    else if (op == '-')
      operandStack.push(op2 - op1);
```

process *

```
    else if (op == '*')
      operandStack.push(op2 * op1);
```

process /

```
    else if (op == '/')
```

```
102      operandStack.push(op2 / op1);
103    }
104
105    public static String insertBlanks(String s) {
106      String result = "";
107
108      for (int i = 0; i < s.length(); i++) {
109        if (s.charAt(i) == '(' || s.charAt(i) == ')' ||
110            s.charAt(i) == '+' || s.charAt(i) == '-' ||
111            s.charAt(i) == '*' || s.charAt(i) == '/')
112          result += " " + s.charAt(i) + " ";
113        else
114          result += s.charAt(i);
115      }
116
117      return result;
118    }
119  }
```

insert blanks

You can use the `GenericStack` class provided by the book, or the `java.util.Stack` class defined in the Java API for creating stacks. This example uses the `java.util.Stack` class. The program will work if it is replaced by `GenericStack`.

The program takes an expression as a command-line argument in one string.

The `evaluateExpression` method creates two stacks, `operandStack` and `operatorStack` (lines 23 and 26), and extracts operands, operators, and parentheses delimited by space (lines 29–32). The `insertBlanks` method is used to ensure that operands, operators, and parentheses are separated by at least one blank (line 29).

The program scans each token in the `for` loop (lines 35–77). If a token is empty, skip it (line 37). If a token is an operand, push it to `operandStack` (line 75). If a token is a `+` or `-` operator (line 38), process all the operators from the top of `operatorStack`, if any (lines 40–46), and push the newly scanned operator into the stack (line 49). If a token is a `*` or `/` operator (line 51), process all the `*` and `/` operators from the top of `operatorStack`, if any (lines 53–57), and push the newly scanned operator to the stack (line 60). If a token is a `(` symbol (line 62), push it into `operatorStack`. If a token is a `)` symbol (line 65), process all the operators from the top of `operatorStack` until seeing the `)` symbol (lines 67–69) and pop the `)` symbol from the stack.

After all tokens are considered, the program processes the remaining operators in `operatorStack` (lines 80–82).

The `processAnOperator` method (lines 90–103) processes an operator. The method pops the operator from `operatorStack` (line 92) and pops two operands from `operandStack` (lines 93 and 94). Depending on the operator, the method performs an operation and pushes the result of the operation back to `operandStack` (lines 96, 98, 100, and 102).

Check Point

20.11.1 Can the `EvaluateExpression` program evaluate the following expressions `"1 + 2"`, `"1 + 2"`, `"(1) + 2"`, `"((1)) + 2"`, and `"(1 + 2)"`?

20.11.2 Show the change of the contents in the stacks when evaluating `"3 + (4 + 5) * (3 + 5) + 4 * 5"` using the `EvaluateExpression` program.

20.11.3 If you enter an expression `"4 + 5 5 5"`, the program will display 10. How do you fix this problem?

KEY TERMS

collection	776
comparator	787
convenience abstract class	777
data structure	776
linked list	784
list	776
priority queue	776
queue	776

Chapter Summary

1. The `Collection` interface defines the common operations for lists, vectors, stacks, queues, priority queues, and sets.

2. Each collection is `Iterable`. You can obtain its `Iterator` object to traverse all the elements in the collection.

3. All the concrete classes except `PriorityQueue` in the Java Collections Framework implement the `Cloneable` and `Serializable` interfaces. Thus, their instances can be cloned and serialized.

4. A list stores an ordered collection of elements. To allow duplicate elements to be stored in a collection, you need to use a list. A list not only can store duplicate elements but also allows the user to specify where they are stored. The user can access elements by an index.

5. Two types of lists are supported: `ArrayList` and `LinkedList`. `ArrayList` is a resizable-array implementation of the `List` interface. All the methods in `ArrayList` are defined in `List`. `LinkedList` is a *linked-list* implementation of the `List` interface. In addition to implementing the `List` interface, this class provides the methods for retrieving, inserting, and removing elements from both ends of the list.

6. `Comparator` can be used to compare the objects of a class that doesn't implement `Comparable`.

7. The `Vector` class extends the `AbstractList` class. Starting with Java 2, `Vector` has been the same as `ArrayList`, except that the methods for accessing and modifying the vector are synchronized. The `Stack` class extends the `Vector` class and provides several methods for manipulating the stack.

8. The `Queue` interface represents a queue. The `PriorityQueue` class implements `Queue` for a *priority queue*.

Quiz

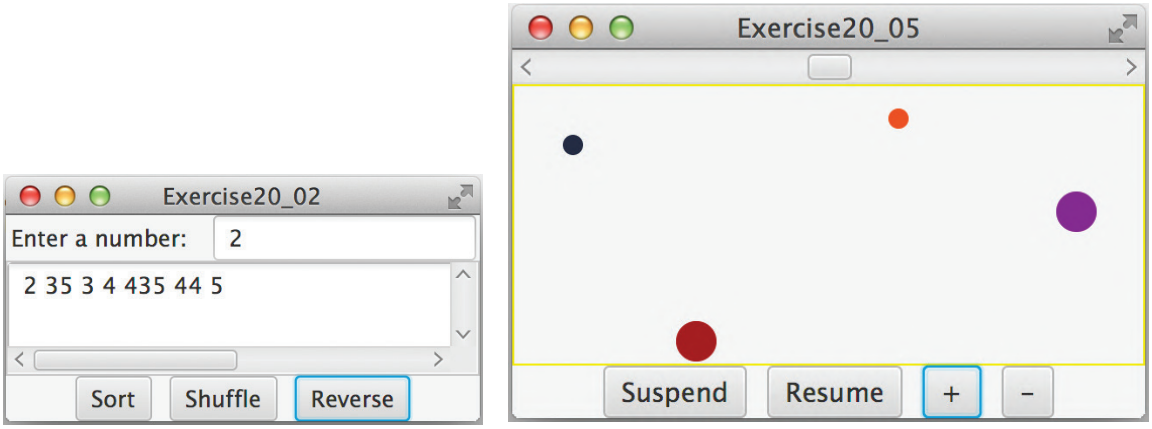
Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

Sections 20.2–20.7

***20.1** (*Display words in ascending alphabetical order*) Write a program that reads words from a text file and displays all the words (duplicates allowed) in ascending alphabetical order. The words must start with a letter. The text file is passed as a command-line argument.

***20.2** (*Store numbers in a linked list*) Write a program that lets the user enter numbers from a graphical user interface and displays them in a text area, as shown in Figure 20.17a. Use a linked list to store the numbers. Do not store duplicate numbers. Add the buttons *Sort*, *Shuffle*, and *Reverse* to sort, shuffle, and reverse the list.



Enter a number: 2

2 35 3 4 435 44 5

(a)

(b)

Figure 20.17 (a) The numbers are stored in a list and displayed in the text area. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) The colliding balls are combined.

***20.3** (*Guessing the capitals*) Rewrite Programming Exercise 8.37 to store the pairs of states and capitals so that the questions are displayed randomly.

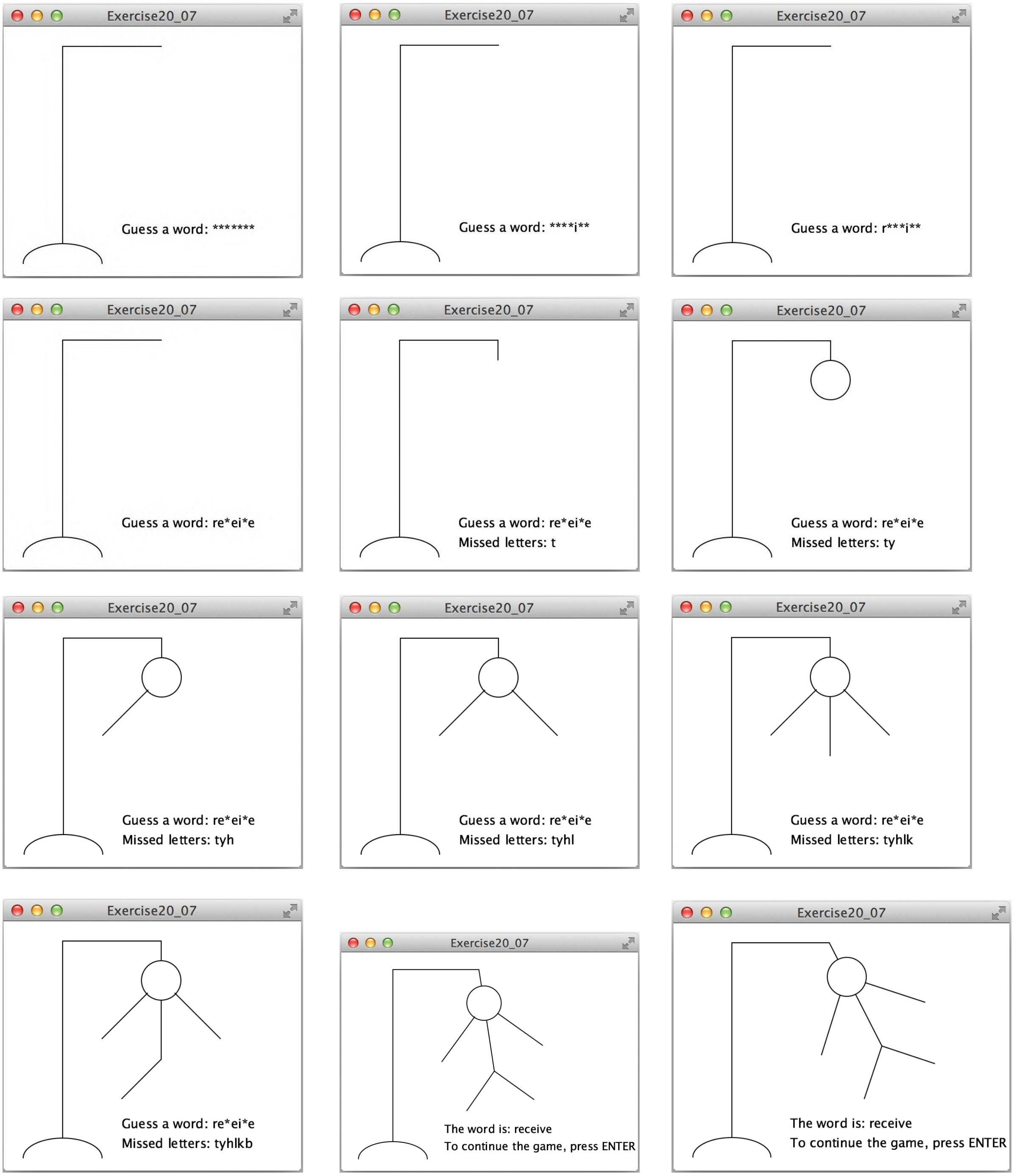
***20.4** (*Sort points in a plane*) Write a program that meets the following requirements. Randomly create `100` points using `Point2D` and apply the `Arrays.sort(list, Comparator)` method to sort the points in increasing order of their *y*-coordinates and then in increasing order of their *x*-coordinates. Display the *x*- and *y*-coordinates of the first five points.

*****20.5** (*Combine colliding bouncing balls*) The example in Section 20.8 displays multiple bouncing balls. Extend the example to detect collisions. Once two balls collide, remove the later ball that was added to the pane and add its radius to the other ball, as shown in Figure 20.17b. Use the *Suspend* button to suspend the animation, and the *Resume* button to resume the animation. Add a mouse-pressed handler that removes a ball when the mouse is pressed on the ball.

20.6 (*Use iterators on linked lists*) Write a test program that stores 5 million integers in a linked list and test the time to traverse the list using an `iterator` vs. using the `get(index)` method.

*****20.7** (*Game: hangman*) Programming Exercise 7.35 presents a console version of the popular hangman game. Write a GUI program that lets a user play the game. The user guesses a word by entering one letter at a time, as shown in Figure 20.18. If the user misses seven times, a hanging man swings. Once a word is finished, the user can press the *Enter* key to continue to guess another word.

****20.8** (*Game: lottery*) Revise Programming Exercise 3.15 to add an additional $2,000 award if two digits from the user input are in the lottery number. (*Hint*: Sort the three digits in the lottery number and three digits in the user input into two lists, and use the `Collection`'s `containsAll` method to check whether the two digits in the user input are in the lottery number.)



```
Guess a word: *******
```

```
Guess a word: ****i**
```

```
Guess a word: r***i**
```

```
Guess a word: re*ei*e
```

```
Guess a word: re*ei*e
Missed letters: t
```

```
Guess a word: re*ei*e
Missed letters: ty
```

```
Guess a word: re*ei*e
Missed letters: tyh
```

```
Guess a word: re*ei*e
Missed letters: tyhl
```

```
Guess a word: re*ei*e
Missed letters: tyhlk
```

```
Guess a word: re*ei*e
Missed letters: tyhlkb
```

```
The word is: receive
To continue the game, press ENTER
```

```
The word is: receive
To continue the game, press ENTER
```

FIGURE 20.18 The program displays a hangman game. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Sections 20.8–20.10

*****20.9** (*Remove the largest ball first*) Modify Listing 20.10, `MultipleBallApp.java` to assign a random radius between 2 and 20 when a ball is created. When the − button is clicked, one of largest balls is removed.

20.10 (*Perform set operations on priority queues*) Write a program that creates two priority queues, {`"George"`, `"Jim"`, `"John"`, `"Blake"`, `"Kevin"`, `"Michael"`} and {`"George"`, `"Katie"`, `"Kevin"`, `"Michelle"`, `"Ryan"`} and displays their union, difference, and intersection.

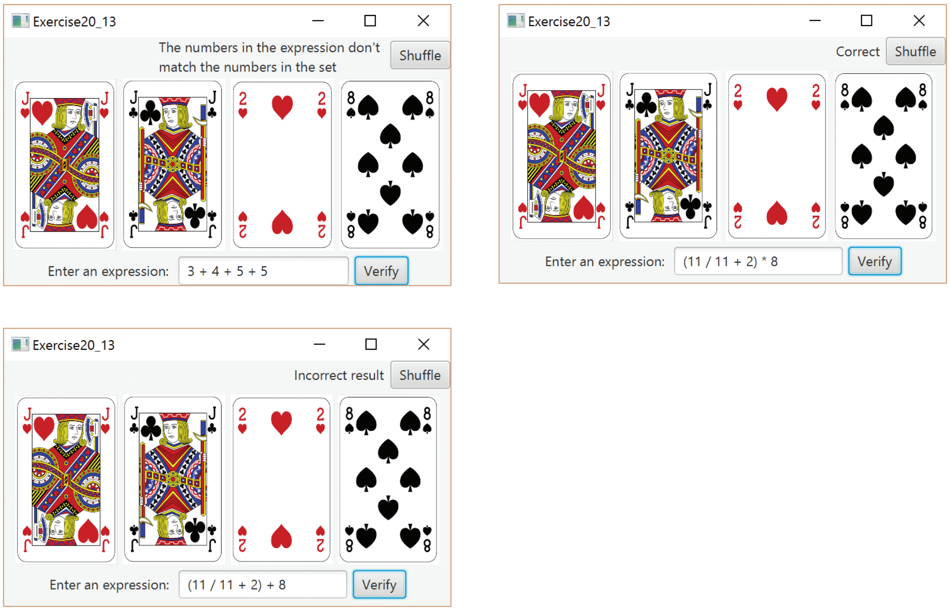
***20.11** (*Match grouping symbols*) A Java program contains various pairs of grouping symbols, such as:

- Parentheses: `(` and `)`
- Braces: `{` and `}`
- Brackets: `[` and `]`

Note the grouping symbols cannot overlap. For example, `(a{b)}` is illegal. Write a program to check whether a Java source-code file has correct pairs of grouping symbols. Pass the source-code file name as a command-line argument.

20.12 (*Clone* `PriorityQueue`) Define `MyPriorityQueue` class that extends `PriorityQueue` to implement the `Cloneable` interface and implement the `clone()` method to clone a priority queue.

****20.13** (*Game: the 24-point card game*) The 24-point card game is to pick any four cards from 52 cards, as shown in Figure 20.19. Note the Jokers are excluded. Each card represents a number. An Ace, King, Queen, and Jack represent `1`, `13`, `12`, and `11`, respectively. You can click the *Shuffle* button to get four new cards. Enter an expression that uses the four numbers from the four selected cards. Each number must be used once and only once. You can use the operators (addition, subtraction, multiplication, and division) and parentheses in the expression. The expression must evaluate to `24`. After entering the expression, click the *Verify* button to check whether the numbers in the expression are currently selected and whether the result of the expression is correct. Display the verification in a label before the *Shuffle* button. Assume that images are stored in files named **1.png**, **2.png**, . . . , **52.png**, in the order of spades, hearts, diamonds, and clubs. Thus, the first 13 images are for spades 1, 2, 3, . . . , and 13.



The numbers in the expression don't match the numbers in the set

Enter an expression: 3 + 4 + 5 + 5

Correct

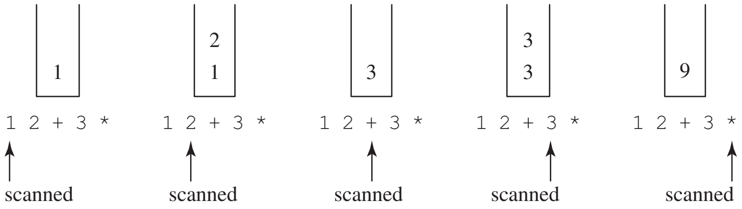
Enter an expression: (11 / 11 + 2) * 8

Incorrect result

Enter an expression: (11 / 11 + 2) + 8

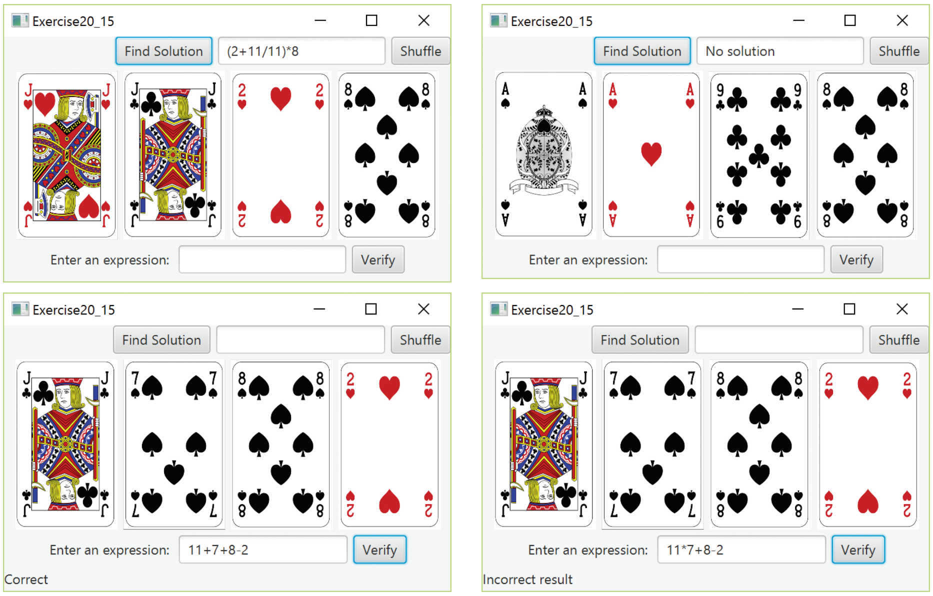
FIGURE 20.19 The user enters an expression consisting of the numbers in the cards and clicks the Verify button to check the answer. *Source*: Fotolia.

****20.14** (*Postfix notation*) Postfix notation is a way of writing expressions without using parentheses. For example, the expression `(1 + 2) * 3` would be written as `1 2 + 3 *`. A postfix expression is evaluated using a stack. Scan a postfix expression from left to right. A variable or constant is pushed into the stack. When an operator is encountered, apply the operator with the top two operands in the stack and replace the two operands with the result. The following diagram shows how to evaluate `1 2 + 3 *`:


1
1 2 + 3 *
scanned
2
1
1 2 + 3 *
scanned
3
1 2 + 3 *
scanned
3
3
1 2 + 3 *
scanned
9
1 2 + 3 *
scanned

Write a program to evaluate postfix expressions. Pass the expression as a command-line argument in one string.

*****20.15** (*Game: the 24-point card game*) Improve Programming Exercise 20.13 to enable the computer to display the expression if one exists, as shown in Figure 20.20. Otherwise, report that the expression does not exist. Place the label for verification result at the bottom of the UI. The expression must use all four cards and evaluated to 24.


Exercise20_15
Find Solution
Shuffle
Enter an expression:
Verify
Exercise20_15
Find Solution
Shuffle
Enter an expression:
Verify
Exercise20_15
Find Solution
Shuffle
Enter an expression:
Verify
Exercise20_15
Find Solution
Shuffle
Enter an expression:
Verify

(2+11/11)*8

No solution

Enter an expression: 11+7+8-2

Correct

Enter an expression: 11*7+8-2

Incorrect result

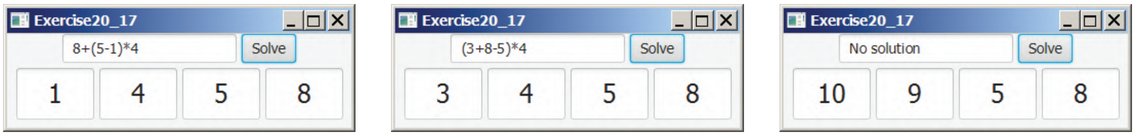
FIGURE 20.20 The program can automatically find a solution if one exists. *Source*: Fotolia.

****20.16** (*Convert infix to postfix*) Write a method that converts an infix expression into a postfix expression using the following header:

```
public static String infixToPostfix(String expression)
```

For example, the method should convert the infix expression `(1 + 2) * 3` to `1 2 + 3 *` and `2 * (1 + 3)` to `2 1 3 + *`. Write a program that accepts an expression in one argument from the command line and displays its corresponding postfix expression.

*****20.17** (*Game: the 24-point card game*) This exercise is a variation of the 24-point card game described in Programming Exercise 20.13. Write a program to check whether there is a 24-point solution for the four specified numbers. The program lets the user enter four values, each between `1` and `13`, as shown in Figure 20.21. The user can then click the *Solve* button to display the solution or display "No solution" if none exists:



8+(5-1)*4

1 4 5 8

(3+8-5)*4

3 4 5 8

No solution

10 9 5 8

FIGURE 20.21 The user enters four numbers and the program finds a solution. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***20.18** (*Directory size*) Listing 18.7, DirectorySize.java, gives a recursive method for finding a directory size. Rewrite this method without using recursion. Your program should use a queue to store the subdirectories under a directory. The algorithm can be described as follows:

```
long getSize(File directory) {
  long size = 0;
  add directory to the queue;

  while (queue is not empty) {
    Remove an item from the queue into t;
    if (t is a file)
      size += t.length();
    else
      add all the files and subdirectories under t into the
        queue;
  }

  return size;
}
```

*****20.19** (*Game: solution ratio for 24-point card game*) When you pick four cards from a deck of 52 cards for the 24-point card game introduced in Programming Exercise 20.13, the four cards may not have a 24-point solution. What is the number of all possible picks of four cards from 52 cards? Among all possible picks, how many of them have 24-point solutions? What is the success ratio—that is, (number of picks with solutions)/(number of all possible picks of four cards)? Write a program to find these answers.

***20.20** (*Directory size*) Rewrite Programming Exercise 18.28 using a stack instead of a queue.

***20.21** (*Use `Comparator`*) Write the following generic method using selection sort and a comparator:

```
public static <E> void selectionSort(E[] list,
  Comparator<? super E> comparator)
```

Write a test program that prompts the user to enter six strings, invokes the sort method to sort the six strings by their last character, and displays the sorted strings. Use `Scanner`'s `next()` method to read a string.

***20.22** (*Nonrecursive Tower of Hanoi*) Implement the `moveDisks` method in Listing 18.8 using a stack instead of using recursion.

****20.23** (*Evaluate expression*) Modify Listing 20.12, EvaluateExpression.java, to add operators `^` for exponent and `%` for remainder. For example, `3 ^ 2` is `9` and `3 % 2` is `1`. The `^` operator has the highest precedence and the `%` operator has the same precedence as the `*` and `/` operators. Your program should prompt the user to enter an expression. Here is a sample run of the program:

```
Enter an expression: (5 * 2 ^ 3 + 2 * 3 % 2) * 4 ↵Enter
(5 * 2 ^ 3 + 2 * 3 % 2) * 4 = 160
```

CHAPTER

21

SETS AND MAPS

Objectives

- To store unordered, nonduplicate elements using a set (§21.2).
- To explore how and when to use **`HashSet`** (§21.2.1), **`LinkedHashSet`** (§21.2.2), or **`TreeSet`** (§21.2.3) to store a set of elements.
- To compare the performance of sets and lists (§21.3).
- To use sets to develop a program that counts the keywords in a Java source file (§21.4).
- To tell the differences between **`Collection`** and **`Map`** and describe when and how to use **`HashMap`**, **`LinkedHashMap`**, or **`TreeMap`** to store values associated with keys (§21.5).
- To use maps to develop a program that counts the occurrence of the words in a text (§21.6).
- To obtain singleton sets, lists, and maps and unmodifiable sets, lists, and maps, use the static methods in the **`Collections`** class (§21.7).

21.1 Introduction

Key Point

A set is an efficient data structure for storing and processing nonduplicate elements. A map is like a dictionary that provides a quick lookup to retrieve a value using a key.

why set?

The "**No-Fly**" **list** is a list, created and maintained by the U.S. government's Terrorist Screening Center, of people who are not permitted to board a commercial aircraft for travel in or out of the United States. Suppose we need to write a program that checks whether a person is on the No-Fly list. You can use a list to store names in the No-Fly list. However, a more efficient data structure for this application is a *set*.

why map?

Suppose your program also needs to store detailed information about terrorists in the No-Fly list. The detailed information such as gender, height, weight, and nationality can be retrieved using the name as the key. A *map* is an efficient data structure for such a task.

This chapter introduces sets and maps in the Java Collections Framework.

21.2 Sets

Key Point

You can create a set using one of its three concrete classes: `HashSet`, `LinkedHashSet`, *or* `TreeSet`.

`set`
no duplicates

The `Set` interface extends the `Collection` interface, as shown in Figure 20.1. It does not introduce new methods or constants, but it stipulates that an instance of `Set` contains no duplicate elements. The concrete classes that implement `Set` must ensure that no duplicate elements can be added to the set.

`AbstractSet`

The `AbstractSet` class extends `AbstractCollection` and partially implements `Set`. The `AbstractSet` class provides concrete implementations for the `equals` method and the `hashCode` method. The hash code of a set is the sum of the hash codes of all the elements in the set. Since the `size` method and `iterator` method are not implemented in the `AbstractSet` class, `AbstractSet` is an abstract class.

Three concrete classes of `Set` are `HashSet`, `LinkedHashSet`, and `TreeSet`, as shown in Figure 21.1.

21.2.1 HashSet

`hash set`

The `HashSet` class is a concrete class that implements `Set`. You can create an empty *hash set* using its no-arg constructor, or create a hash set from an existing collection. By default, the initial capacity is `16` and the load factor is `0.75`. If you know the size of your set, you can specify the initial capacity and load factor in the constructor. Otherwise, use the default setting. The load factor is a value between `0.0` and `1.0`.

load factor

The load factor measures how full the set is allowed to be before its capacity is increased. When the number of elements exceeds the product of the capacity and load factor, the capacity is automatically doubled. For example, if the capacity is `16` and load factor is `0.75`, the capacity will be doubled to `32` when the size reaches `12` ($16 * 0.75 = 12$). A higher load factor decreases the space costs but increases the search time. Generally, the default load factor `0.75` is a good trade-off between time and space costs. We will discuss more on the load factor in Chapter 27, Hashing.

`hashCode()`

A `HashSet` can be used to store *duplicate-free* elements. For efficiency, objects added to a hash set need to implement the `hashCode` method in a manner that properly disperses the hash code. The `hashCode` method is defined in the `Object` class. The hash codes of two objects must be the same if the two objects are equal. Two unequal objects may have the same hash code, but you should implement the `hashCode` method to avoid too many such cases. Most of the classes in the Java API implement the `hashCode` method. For example, the `hashCode` in the `Integer` class returns its `int` value. The `hashCode` in the `Character` class returns the Unicode of the character. The `hashCode` in the `String` class returns $s_0 * 31^{(n-1)} + s_1 * 31^{(n-2)} + \cdots + s_{n-1}$, where s_i is `s.charAt(i)`.

`Set` does not store duplicate elements. Two elements `e1` and `e2` are considered duplicate for a `HashSet` if `e1.equals(e2)` is true and `e1.hashCode() == e2.hashCode()`. Note that

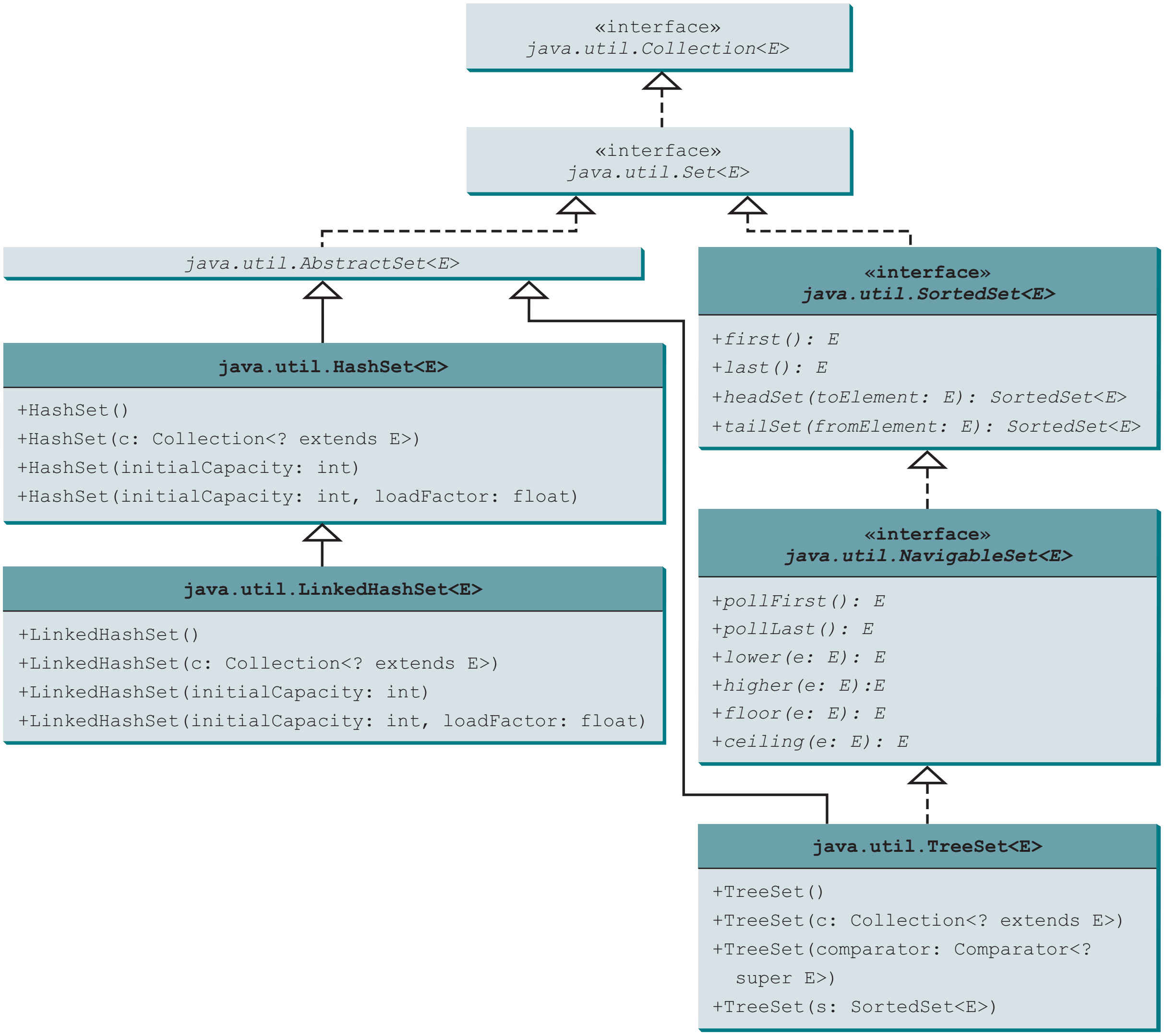

«interface»
java.util.Collection<E>
«interface»
java.util.Set<E>
java.util.AbstractSet<E>
«interface»
java.util.SortedSet<E>
+first(): E
+last(): E
+headSet(toElement: E): SortedSet<E>
+tailSet(fromElement: E): SortedSet<E>
java.util.HashSet<E>
+HashSet()
+HashSet(c: Collection<? extends E>)
+HashSet(initialCapacity: int)
+HashSet(initialCapacity: int, loadFactor: float)
java.util.LinkedHashSet<E>
+LinkedHashSet()
+LinkedHashSet(c: Collection<? extends E>)
+LinkedHashSet(initialCapacity: int)
+LinkedHashSet(initialCapacity: int, loadFactor: float)
«interface»
java.util.NavigableSet<E>
+pollFirst(): E
+pollLast(): E
+lower(e: E): E
+higher(e: E):E
+floor(e: E): E
+ceiling(e: E): E
java.util.TreeSet<E>
+TreeSet()
+TreeSet(c: Collection<? extends E>)
+TreeSet(comparator: Comparator<? super E>)
+TreeSet(s: SortedSet<E>)

FIGURE 21.1 The Java Collections Framework provides three concrete set classes.

by contract, if two elements are equal, their `hashCode` must be same. So you need to override the `hashCode()` method whenever the `equals` method is overridden in the class.

Listing 21.1 gives a program that creates a hash set to store strings and uses a foreach loop and a `forEach` method to traverse the elements in the set.

Listing 21.1 gives a program that creates a hash set to store strings and uses a foreach loop and a `forEach` method to traverse the elements in the set.

LISTING 21.1 TestHashSet.java

```java
import java.util.*;

public class TestHashSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<>();

    // Add strings to the set
    set.add("London");
```

create a set

add element

traverse elements

forEach method

```
10      set.add("Paris");
11      set.add("New York");
12      set.add("San Francisco");
13      set.add("Beijing");
14      set.add("New York");
15
16      System.out.println(set);
17
18      // Display the elements in the hash set
19      for (String s: set) {
20        System.out.print(s.toUpperCase() + " ");
21      }
22
23      // Process the elements using a forEach method
24      System.out.println();
25      set.forEach(e -> System.out.print(e.toLowerCase() + " "));
26    }
27  }
```

```
[San Francisco, New York, Paris, Beijing, London]
SAN FRANCISCO NEW YORK PARIS BEIJING LONDON
```

The strings are added to the set (lines 9–14). `New York` is added to the set more than once, but only one string is stored because a set does not allow duplicates.

As shown in the output, the strings are not stored in the order in which they are inserted into the set. There is no particular order for the elements in a hash set. To impose an order on them, you need to use the `LinkedHashSet` class, which is introduced in the next section.

Recall that the `Collection` interface extends the `Iterable` interface, so the elements in a set are iterable. A foreach loop is used to traverse all the elements in the set (lines 19–21). You can also use a `forEach` method to process each element in a set (line 25).

Since a set is an instance of `Collection`, all methods defined in `Collection` can be used for sets. Listing 21.2 gives an example that applies the methods in the `Collection` interface on sets.

LISTING 21.2 TestMethodsInCollection.java

create a set

add element

get size

remove element

create a set

```
public class TestMethodsInCollection {
  public static void main(String[] args) {
    // Create set1
    java.util.Set<String> set1 = new java.util.HashSet<>();

    // Add strings to set1
    set1.add("London");
    set1.add("Paris");
    set1.add("New York");
    set1.add("San Francisco");
    set1.add("Beijing");

    System.out.println("set1 is " + set1);
    System.out.println(set1.size() + " elements in set1");

    // Delete a string from set1
    set1.remove("London");
    System.out.println("\nset1 is " + set1);
    System.out.println(set1.size() + " elements in set1");

    // Create set2
    java.util.Set<String> set2 = new java.util.HashSet<>();

    // Add strings to set2
25      set2.add("London");
26      set2.add("Shanghai");
27      set2.add("Paris");
28      System.out.println("\nset2 is " + set2);
29      System.out.println(set2.size() + " elements in set2");
30
31      System.out.println("\nIs Taipei in set2? "
32        + set2.contains("Taipei"));
33
34      set1.addAll(set2);
35      System.out.println("\nAfter adding set2 to set1, set1 is "
36        + set1);
37
38      set1.removeAll(set2);
39      System.out.println("After removing set2 from set1, set1 is "
40        + set1);
41
42      set1.retainAll(set2);
43      System.out.println("After retaining common elements in set2 "
44        + "and set2, set1 is " + set1);
45    }
46  }
```

add element

contains element?

`addAll`

`removeAll`

`retainAll`

```
set1 is [San Francisco, New York, Paris, Beijing, London]

5 elements in set1

set1 is [San Francisco, New York, Paris, Beijing]
4 elements in set1

set2 is [Shanghai, Paris, London]
3 elements in set2

Is Taipei in set2? false

After adding set2 to set1, set1 is
  [San Francisco, New York, Shanghai, Paris, Beijing, London]

After removing set2 from set1, set1 is
  [San Francisco, New York, Beijing]

After retaining common elements in set1 and set2, set1 is []
```

The program creates two sets (lines 4 and 22). The `size()` method returns the number of the elements in a set (line 14). Line 17

```
set1.remove("London");
```

removes `London` from `set1`.

The `contains` method (line 32) checks whether an element is in the set.

Line 34

```
set1.addAll(set2);
```

adds `set2` to `set1`. Therefore, `set1` becomes `[San Francisco, New York, Shanghai, Paris, Beijing, London]`.

Line 38

```
set1.removeAll(set2);
```

removes **`set2`** from **`set1`**. Thus, **`set1`** becomes **`[San Francisco, New York, Beijing]`**.

Line 42

```
set1.retainAll(set2);
```

retains the common elements in **`set1`** and **`set2`**. Since **`set1`** and **`set2`** have no common elements, **`set1`** becomes empty.

21.2.2 LinkedHashSet

LinkedHashSet

`LinkedHashSet` extends **`HashSet`** with a linked-list implementation that supports an ordering of the elements in the set. The elements in a **`HashSet`** are not ordered, but the elements in a **`LinkedHashSet`** can be retrieved in the order in which they were inserted into the set. A **`LinkedHashSet`** can be created by using one of its four constructors, as shown in Figure 21.1. These constructors are similar to the constructors for **`HashSet`**.

Listing 21.3 gives a test program for **`LinkedHashSet`**. The program simply replaces **`HashSet`** by **`LinkedHashSet`** in Listing 21.1.

LISTING 21.3 TestLinkedHashSet.java

create linked hash set

add element

display elements

```
 1  import java.util.*;
 2
 3  public class TestLinkedHashSet {
 4    public static void main(String[] args) {
 5      // Create a hash set
 6      Set<String> set = new LinkedHashSet<>();
 7
 8      // Add strings to the set
 9      set.add("London");
10      set.add("Paris");
11      set.add("New York");
12      set.add("San Francisco");
13      set.add("Beijing");
14      set.add("New York");
15
16      System.out.println(set);
17
18      // Display the elements in the hash set
19      for (String element: set)
20        System.out.print(element.toLowerCase() + " ");
21    }
22  }
```

```
[London, Paris, New York, San Francisco, Beijing]
london paris new york san francisco beijing
```

A **`LinkedHashSet`** is created in line 6. As shown in the output, the strings are stored in the order in which they are inserted. Since **`LinkedHashSet`** is a set, it does not store duplicate elements.

The **`LinkedHashSet`** maintains the order in which the elements are inserted. To impose a different order (e.g., increasing or decreasing order), you can use the **`TreeSet`** class, which is introduced in the next section.

Tip

If you don't need to maintain the order in which the elements are inserted, use **`HashSet`**, which is more efficient than **`LinkedHashSet`**.

21.2.3 `TreeSet`

As shown in Figure 21.1, `SortedSet` is a subinterface of `Set`, which guarantees that the elements in the set are sorted. In addition, it provides the methods `first()` and `last()` for returning the first and last elements in the set, and `headSet(toElement)` and `tailSet(fromElement)` for returning a portion of the set whose elements are less than `toElement` and greater than or equal to `fromElement`, respectively.

`NavigableSet` extends `SortedSet` to provide navigation methods `lower(e)`, `floor(e)`, `ceiling(e)`, and `higher(e)` that return elements, respectively, less than, less than or equal, greater than or equal, and greater than a given element and return `null` if there is no such element. The `pollFirst()` and `pollLast()` methods remove and return the first and last element in the tree set, respectively.

`TreeSet` implements the `SortedSet` interface. To create a `TreeSet`, use a constructor, as shown in Figure 21.1. You can add objects into a *tree set* as long as they can be compared with each other.

tree set

As discussed in Section 20.5, the elements can be compared in two ways: using the `Comparable` interface or the `Comparator` interface.

Listing 21.4 gives an example of ordering elements using the `Comparable` interface. The preceding example in Listing 21.3 displays all the strings in their insertion order. This example rewrites the preceding example to display the strings in alphabetical order using the `TreeSet` class.

Listing 21.4 `TestTreeSet.java`

```java
import java.util.*;

public class TestTreeSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<>();

    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");

    TreeSet<String> treeSet = new TreeSet<>(set);
    System.out.println("Sorted tree set: " + treeSet);

    // Use the methods in SortedSet interface
    System.out.println("first(): " + treeSet.first());
    System.out.println("last(): " + treeSet.last());
    System.out.println("headSet(\"New York\"): " +
      treeSet.headSet("New York"));
    System.out.println("tailSet(\"New York\"): " +
      treeSet.tailSet("New York"));

    // Use the methods in NavigableSet interface
    System.out.println("lower(\"P\"): " + treeSet.lower("P"));
    System.out.println("higher(\"P\"): " + treeSet.higher("P"));
    System.out.println("floor(\"P\"): " + treeSet.floor("P"));
    System.out.println("ceiling(\"P\"): " + treeSet.ceiling("P"));
    System.out.println("pollFirst(): " + treeSet.pollFirst());
    System.out.println("pollLast(): " + treeSet.pollLast());
    System.out.println("New tree set: " + treeSet);
  }
}
```

create hash set

create tree set

display elements

```
Sorted tree set: [Beijing, London, New York, Paris, San Francisco]
first(): Beijing
last(): San Francisco
headSet("New York"): [Beijing, London]
tailSet("New York"): [New York, Paris, San Francisco]
lower("P"): New York
higher("P"): Paris
floor("P"): New York
ceiling("P"): Paris
pollFirst(): Beijing
pollLast(): San Francisco
New tree set: [London, New York, Paris]
```

The example creates a hash set filled with strings, then creates a tree set for the same strings. The strings are sorted in the tree set using the `compareTo` method in the `Comparable` interface. Two elements `e1` and `e2` are considered duplicate for a `TreeSet` if `e1.compareTo(e2)` is `0` for `Comparable` and `e1.compare(e2)` is `0` for `Comparator`.

The elements in the set are sorted once you create a `TreeSet` object from a `HashSet` object using `new TreeSet<>(set)` (line 16). You may rewrite the program to create an instance of `TreeSet` using its no-arg constructor and add the strings into the `TreeSet` object.

`treeSet.first()` returns the first element in `treeSet` (line 20) and `treeSet.last()` returns the last element in `treeSet` (line 21). `treeSet.headSet("New York")` returns the elements in `treeSet` before New York (lines 22–23). `treeSet.tailSet("New York")` returns the elements in `treeSet` after New York, including New York (lines 24–25).

`treeSet.lower("P")` returns the largest element less than `P` in `treeSet` (line 28). `treeSet.higher("P")` returns the smallest element greater than `P` in `treeSet` (line 29). `treeSet.floor("P")` returns the largest element less than or equal to `P` in `treeSet` (line 30). `treeSet.ceiling("P")` returns the smallest element greater than or equal to `P` in `treeSet` (line 31). `treeSet.pollFirst()` removes the first element in `treeSet` and returns the removed element (line 32). `treeSet.pollLast()` removes the last element in `treeSet` and returns the removed element (line 33).

Note

All the concrete classes in Java Collections Framework (see Figure 20.1) have at least two constructors. One is the no-arg constructor that constructs an empty collection. The other constructs instances from a collection. Thus the `TreeSet` class has the constructor `TreeSet(Collection c)` for constructing a `TreeSet` from a collection `c`. In this example, `new TreeSet<>(set)` creates an instance of `TreeSet` from the collection `set`.

Tip

If you don't need to maintain a sorted set when updating a set, you should use a hash set because it takes less time to insert and remove elements in a hash set. When you need a sorted set, you can create a tree set from the hash set.

If you create a `TreeSet` using its no-arg constructor, the `compareTo` method is used to compare the elements in the set, assuming the class of the elements implements the `Comparable` interface. To use a comparator, you have to use the constructor `TreeSet(Comparator comparator)` to create a sorted set that uses the `compare` method in the comparator to order the elements in the set.

Listing 21.5 gives a program that demonstrates how to sort elements in a tree set using the `Comparator` interface.

LISTING 21.5 `TestTreeSetWithComparator.java`

```
1   import java.util.*;
2
3   public class TestTreeSetWithComparator {
4     public static void main(String[] args) {
5       // Create a tree set for geometric objects using a comparator
6       Set<GeometricObject> set =
7         new TreeSet<>(new GeometricObjectComparator());
8       set.add(new Rectangle(4, 5));
9       set.add(new Circle(40));
10      set.add(new Circle(40));
11      set.add(new Rectangle(4, 1));
12
13      // Display geometric objects in the tree set
14      System.out.println("A sorted set of geometric objects");
15      for (GeometricObject element: set)
16        System.out.println("area = " + element.getArea());
17    }
18  }
```

tree set

display elements

```
A sorted set of geometric objects
area = 4.0
area = 20.0
area = 5021.548245743669
```

The `GeometricObjectComparator` class is defined in Listing 20.4. The program creates a tree set of geometric objects using the `GeometricObjectComparator` for comparing the elements in the set (lines 6 and 7).

The `Circle` and `Rectangle` classes were defined in Section 13.2, Abstract Classes. They are all subclasses of `GeometricObject`. They are added to the set (lines 8–11).

Two circles of the same radius are added to the tree set (lines 9 and 10), but only one is stored because the two circles are equal (determined by the comparator in this case) and the set does not allow duplicates.

Check Point

21.2.1 How do you create an instance of `Set`? How do you insert a new element in a set? How do you remove an element from a set? How do you find the size of a set?

21.2.2 If two objects `o1` and `o2` are equal, what is `o1.equals(o2)` and `o1.hashCode() == o2.hashCode()`?

21.2.3 How do you traverse the elements in a set?

21.2.4 Suppose `set1` is a set that contains the strings `red`, `yellow`, and `green` and that `set2` is another set that contains the strings `red`, `yellow`, and `blue`. Answer the following questions:

- What are in `set1` and `set2` after executing `set1.addAll(set2)`?
- What are in `set1` and `set2` after executing `set1.add(set2)`?
- What are in `set1` and `set2` after executing `set1.removeAll(set2)`?
- What are in `set1` and `set2` after executing `set1.remove(set2)`?
- What are in `set1` and `set2` after executing `set1.retainAll(set2)`?
- What is in `set1` after executing `set1.clear()`?

21.2.5 Show the output of the following code:

```
import java.util.*;

public class Test {
  public static void main(String[] args) {
    LinkedHashSet<String> set1 = new LinkedHashSet<>();
    set1.add("New York");
    LinkedHashSet<String> set2 = set1;
    LinkedHashSet<String> set3 =
      (LinkedHashSet<String>)(set1.clone());
    set1.add("Atlanta");
    System.out.println("set1 is " + set1);
    System.out.println("set2 is " + set2);
    System.out.println("set3 is " + set3);
    set1.forEach(e -> System.out.print(e + " "));
  }
}
```

21.2.6 Show the output of the following code:

```
Set<String> set = new LinkedHashSet<>();
set.add("ABC");
set.add("ABD");
System.out.println(set);
```

21.2.7 What are the differences among `HashSet`, `LinkedHashSet`, and `TreeSet`?

21.2.8 How do you sort the elements in a set using the `compareTo` method in the `Comparable` interface? How do you sort the elements in a set using the `Comparator` interface? What would happen if you added an element that could not be compared with the existing elements in a tree set?

21.2.9 What will the output be if lines 6–7 in Listing 21.5 are replaced by the following code:

```
Set<GeometricObject> set = new HashSet<>();
```

21.2.10 Show the output of the following code:

```
Set<String> set = new TreeSet<>(
  Comparator.comparing(String::length));
set.add("ABC");
set.add("ABD");
System.out.println(set);
```

21.3 Comparing the Performance of Sets and Lists

Key Point

Sets are more efficient than lists for storing nonduplicate elements. Lists are useful for accessing elements through the index.

The elements in a list can be accessed through the index. However, sets do not support indexing because the elements in a set are unordered. To traverse all elements in a set, use a foreach loop. We now conduct an interesting experiment to test the performance of sets and lists. Listing 21.6 gives a program that shows the execution time of (1) testing whether an element is in a hash set, linked hash set, tree set, array list, or linked list and (2) removing elements from a hash set, linked hash set, tree set, array list, and linked list.

LISTING 21.6 SetListPerformanceTest.java

```java
import java.util.*;

public class SetListPerformanceTest {
  static final int N = 50000;

  public static void main(String[] args) {
    // Add numbers 0, 1, 2, ..., N - 1 to the array list
    List<Integer> list = new ArrayList<>();
    for (int i = 0; i < N; i++)
      list.add(i);
    Collections.shuffle(list); // Shuffle the array list

    // Create a hash set, and test its performance
    Collection<Integer> set1 = new HashSet<>(list);
    System.out.println("Member test time for hash set is " +
      getTestTime(set1) + " milliseconds");
    System.out.println("Remove element time for hash set is " +
      getRemoveTime(set1) + " milliseconds");

    // Create a linked hash set, and test its performance
    Collection<Integer> set2 = new LinkedHashSet<>(list);
    System.out.println("Member test time for linked hash set is " +
      getTestTime(set2) + " milliseconds");
    System.out.println("Remove element time for linked hash set is "
      + getRemoveTime(set2) + " milliseconds");

    // Create a tree set, and test its performance
    Collection<Integer> set3 = new TreeSet<>(list);
    System.out.println("Member test time for tree set is " +
      getTestTime(set3) + " milliseconds");
    System.out.println("Remove element time for tree set is " +
      getRemoveTime(set3) + " milliseconds");

    // Create an array list, and test its performance
    Collection<Integer> list1 = new ArrayList<>(list);
    System.out.println("Member test time for array list is " +
      getTestTime(list1) + " milliseconds");
    System.out.println("Remove element time for array list is " +
      getRemoveTime(list1) + " milliseconds");

    // Create a linked list, and test its performance
    Collection<Integer> list2 = new LinkedList<>(list);
    System.out.println("Member test time for linked list is " +
      getTestTime(list2) + " milliseconds");
    System.out.println("Remove element time for linked list is " +
      getRemoveTime(list2) + " milliseconds");
  }

  public static long getTestTime(Collection<Integer> c) {
    long startTime = System.currentTimeMillis();

    // Test if a number is in the collection
    for (int i = 0; i < N; i++)
      c.contains((int)(Math.random() * 2 * N));

    return System.currentTimeMillis() - startTime;
```

create test data

shuffle

a hash set

a linked hash set

a tree set

an array list

a linked list

start time

test membership

return execution time

remove from container

return execution time

```
57    }
58
59    public static long getRemoveTime(Collection<Integer> c) {
60      long startTime = System.currentTimeMillis();
61
62      for (int i = 0; i < N; i++)
63        c.remove(i);
64
65      return System.currentTimeMillis() - startTime;
66    }
67  }
```

```
Member test time for hash set is 20 milliseconds
Remove element time for hash set is 27 milliseconds
Member test time for linked hash set is 27 milliseconds
Remove element time for linked hash set is 26 milliseconds
Member test time for tree set is 47 milliseconds
Remove element time for tree set is 34 milliseconds
Member test time for array list is 39802 milliseconds
Remove element time for array list is 16196 milliseconds
Member test time for linked list is 52197 milliseconds
Remove element time for linked list is 14870 milliseconds
```

The program creates a list for numbers from `0` to `N-1` (for `N` = `50000`) (lines 8–10) and shuffles the list (line 11). From this list, the program creates a hash set (line 14), a linked hash set (line 21), a tree set (line 28), an array list (line 35), and a linked list (line 42). The program obtains the execution time for testing whether a number is in the hash set (line 16), linked hash set (line 23), tree set (line 30), array list (line 37), or linked list (line 44) and obtains the execution time for removing the elements from the hash set (line 18), linked hash set (line 25), tree set (line 32), array list (line 39), and linked list (line 46).

The `getTestTime` method invokes the `contains` method to test whether a number is in the container (line 54) and the `getRemoveTime` method invokes the `remove` method to remove an element from the container (line 63).

sets are better

As these runtimes illustrate, sets are much more efficient than lists for testing whether an element is in a set or a list. Therefore, the No-Fly list should be implemented using a hash set instead of a list, because it is much faster to test whether an element is in a hash set than in a list.

You may wonder why sets are more efficient than lists. The questions will be answered in Chapters 24 and 27 when we introduce the implementations of lists and sets.

Check Point

21.3.1 Suppose you need to write a program that stores unordered, nonduplicate elements, what data structure should you use?

21.3.2 Suppose you need to write a program that stores nonduplicate elements in the order of insertion, what data structure should you use?

21.3.3 Suppose you need to write a program that stores nonduplicate elements in increasing order of the element values, what data structure should you use?

21.3.4 Suppose you need to write a program that stores a fixed number of the elements (possibly duplicates), what data structure should you use?

21.3.5 Suppose you need to write a program that stores the elements in a list with frequent operations to append and delete elements at the end of the list, what data structure should you use?

21.3.6 Suppose you need to write a program that stores the elements in a list with frequent operations to insert and delete elements at the beginning of the list, what data structure should you use?

21.4 Case Study: Counting Keywords

This section presents an application that counts the number of keywords in a Java source file.

Key Point

For each word in a Java source file, we need to determine whether the word is a keyword. To handle this efficiently, store all the keywords in a `HashSet` and use the `contains` method to test if a word is in the keyword set. Listing 21.7 gives this program.

LISTING 21.7 CountKeywords.java

```java
import java.util.*;
import java.io.*;

public class CountKeywords {
  public static void main(String[] args) throws Exception {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a Java source file: ");
    String filename = input.nextLine();

    File file = new File(filename);
    if (file.exists()) {
      System.out.println("The number of keywords in " + filename
        + " is " + countKeywords(file));
    }
    else {
      System.out.println("File " + filename + " does not exist");
    }
  }

  public static int countKeywords(File file) throws Exception {
    // Array of all Java keywords + true, false and null
    String[] keywordString = {"abstract", "assert", "boolean",
        "break", "byte", "case", "catch", "char", "class", "const",
        "continue", "default", "do", "double", "else", "enum",
        "extends", "for", "final", "finally", "float", "goto",
        "if", "implements", "import", "instanceof", "int",
        "interface", "long", "native", "new", "package", "private",
        "protected", "public", "return", "short", "static",
        "strictfp", "super", "switch", "synchronized", "this",
        "throw", "throws", "transient", "try", "void", "volatile",
        "while", "true", "false", "null"};

    Set<String> keywordSet =
      new HashSet<>(Arrays.asList(keywordString));
    int count = 0;

    Scanner input = new Scanner(file);

    while (input.hasNext()) {
      String word = input.next();
      if (keywordSet.contains(word))
        count++;
    }

    return count;
  }
}
```

enter a filename

file exists?

count keywords

keywords

keyword set

is a keyword?

```
Enter a Java source file: c:\ Welcome.java ↵Enter
The number of keywords in c:\ Welcome.java is 5
```

```
Enter a Java source file: c:\ TTT.java ↵Enter
File c:\ TTT.java does not exist
```

The program prompts the user to enter a Java source filename (line 7) and reads the filename (line 8). If the file exists, the `countKeywords` method is invoked to count the keywords in the file (line 13).

The `countKeywords` method creates an array of strings for the keywords (lines 22–31) and creates a hash set from this array (lines 33–34). It then reads each word from the file and tests if the word is in the set (line 41). If so, the program increases the count by `1` (line 42).

You may rewrite the program to use a `LinkedHashSet`, `TreeSet`, `ArrayList`, or `LinkedList` to store the keywords. However, using a `HashSet` is the most efficient for this program.

Check Point

21.4.1 Will the `CountKeywords` program work if lines 33–34 are changed to

```
Set<String> keywordSet =
  new LinkedHashSet<>(Arrays.asList(keywordString));
```

21.4.2 Will the `CountKeywords` program work if lines 33–34 are changed to

```
List<String> keywordSet =
  new ArrayList<>(Arrays.asList(keywordString));
```

21.5 Maps

Key Point

You can create a map using one of its three concrete classes: `HashMap`*,* `LinkedHashMap`*, or* `TreeMap`*.*

map

A *map* is a container object that stores a collection of key/value pairs. It enables fast retrieval, deletion, and updating of the pair through the key. A map stores the values along with the keys. The keys are like indexes. In `List`, the indexes are integers. In `Map`, the keys can be any objects. A map cannot contain duplicate keys. Each key maps to one value. A key and its corresponding value form an entry stored in a map, as shown in Figure 21.2a. Figure 21.2b shows a map in which each entry consists of a Social Security number as the key and a name as the value.

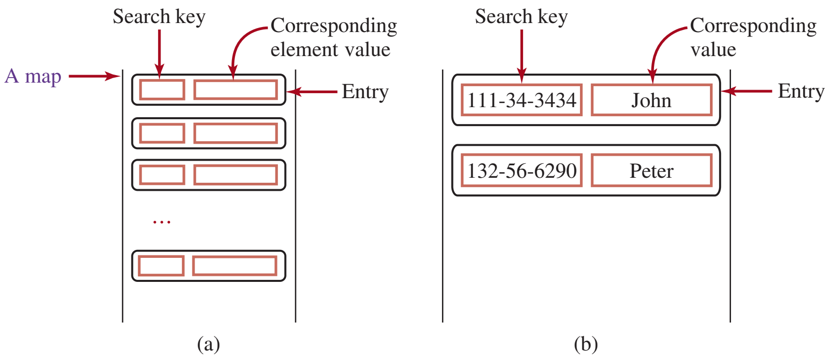


FIGURE 21.2 The entries consisting of key/value pairs are stored in a map.

There are three types of maps: `HashMap`, `LinkedHashMap`, and `TreeMap`. The common features of these maps are defined in the `Map` interface. Their relationship is shown in Figure 21.3.

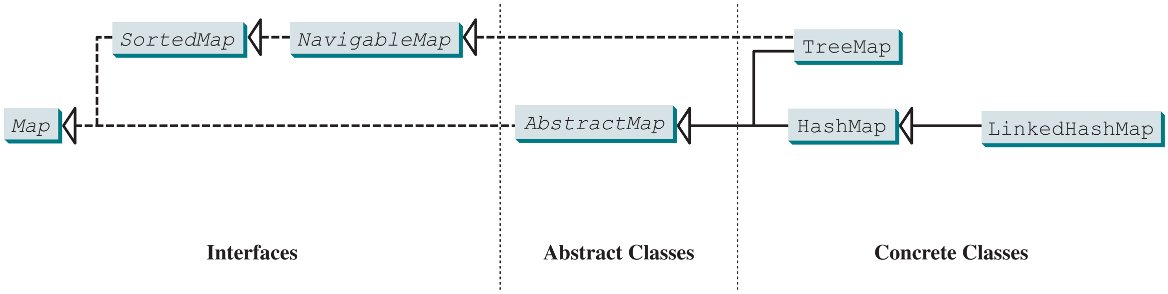
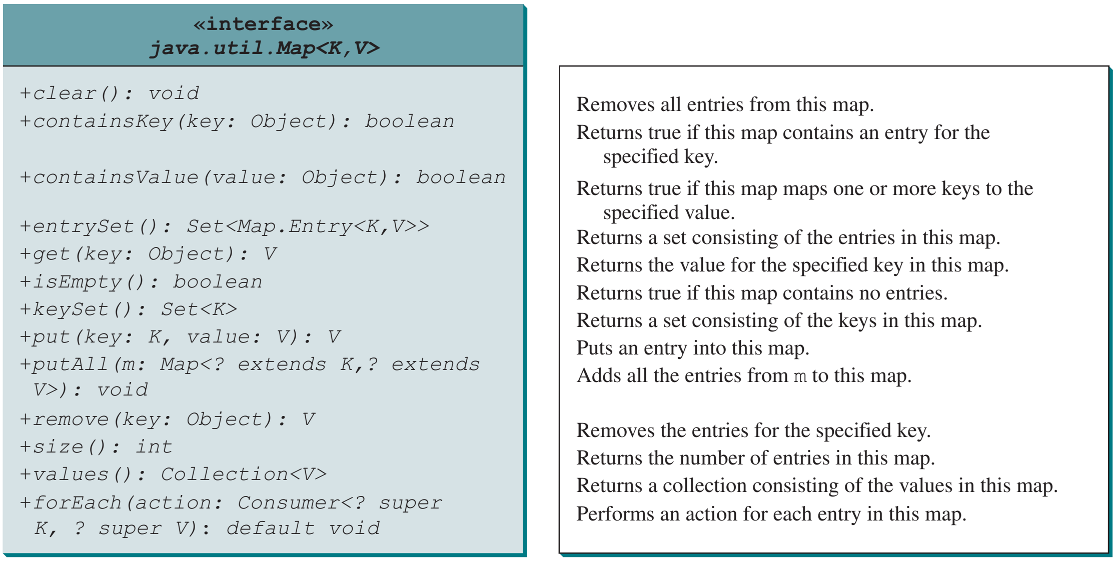


FIGURE 21.3 A map stores key/value pairs.

The `Map` interface provides the methods for querying, updating, and obtaining a collection of values and a set of keys, as shown in Figure 21.4.



«interface»
java.util.Map<K,V>

`+clear(): void`	Removes all entries from this map.
`+containsKey(key: Object): boolean`	Returns true if this map contains an entry for the specified key.
`+containsValue(value: Object): boolean`	Returns true if this map maps one or more keys to the specified value.
`+entrySet(): Set<Map.Entry<K,V>>`	Returns a set consisting of the entries in this map.
`+get(key: Object): V`	Returns the value for the specified key in this map.
`+isEmpty(): boolean`	Returns true if this map contains no entries.
`+keySet(): Set<K>`	Returns a set consisting of the keys in this map.
`+put(key: K, value: V): V`	Puts an entry into this map.
`+putAll(m: Map<? extends K,? extends V>): void`	Adds all the entries from `m` to this map.
`+remove(key: Object): V`	Removes the entries for the specified key.
`+size(): int`	Returns the number of entries in this map.
`+values(): Collection<V>`	Returns a collection consisting of the values in this map.
`+forEach(action: Consumer<? super K, ? super V): default void`	Performs an action for each entry in this map.

FIGURE 21.4 The `Map` interface maps keys to values.

update methods

The *update methods* include `clear`, `put`, `putAll`, and `remove`. The `clear()` method removes all entries from the map. The `put(K key, V value)` method adds an entry for the specified key and value in the map. If the map formerly contained an entry for this key, the old value is replaced by the new value, and the old value associated with the key is returned. The `putAll(Map m)` method adds all entries in `m` to this map. The `remove(Object key)` method removes the entry for the specified key from the map.

query methods

The *query methods* include `containsKey`, `containsValue`, `isEmpty`, and `size`. The `containsKey(Object key)` method checks whether the map contains an entry for the specified key. The `containsValue(Object value)` method checks whether the map contains an entry for this value. The `isEmpty()` method checks whether the map contains any entries. The `size()` method returns the number of entries in the map.

`keySet()`
`values()`
`entrySet()`

You can obtain a set of the keys in the map using the **`keySet()`** method, and a collection of the values in the map using the **`values()`** method. The **`entrySet()`** method returns a set of entries. The entries are instances of the **`Map.Entry<K, V>`** interface, where **`Entry`** is an inner interface for the **`Map`** interface, as shown in Figure 21.5. Each entry in the set is a key/value pair in the underlying map.

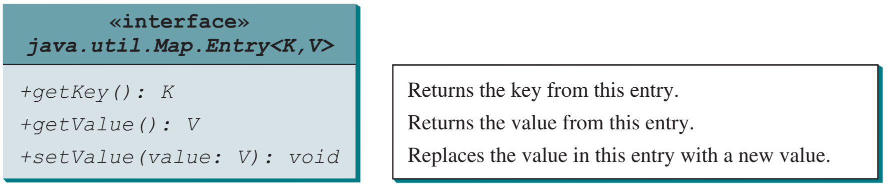

«interface»
java.util.Map.Entry<K,V>
+getKey(): K
+getValue(): V
+setValue(value: V): void
Returns the key from this entry.
Returns the value from this entry.
Replaces the value in this entry with a new value.

FIGURE 21.5 The **`Map.Entry`** interface operates on an entry in the map.

`forEach` method

Java 8 added a default **`forEach`** method in the **`Map`** interface for performing an action on each entry in the map. This method can be used like an iterator for traversing the entries in the map.

`AbstractMap`

The **`AbstractMap`** class is a convenience abstract class that implements all the methods in the **`Map`** interface except the **`entrySet()`** method.

concrete implementation

The **`HashMap`**, **`LinkedHashMap`**, and **`TreeMap`** classes are three *concrete implementations* of the **`Map`** interface, as shown in Figure 21.6.

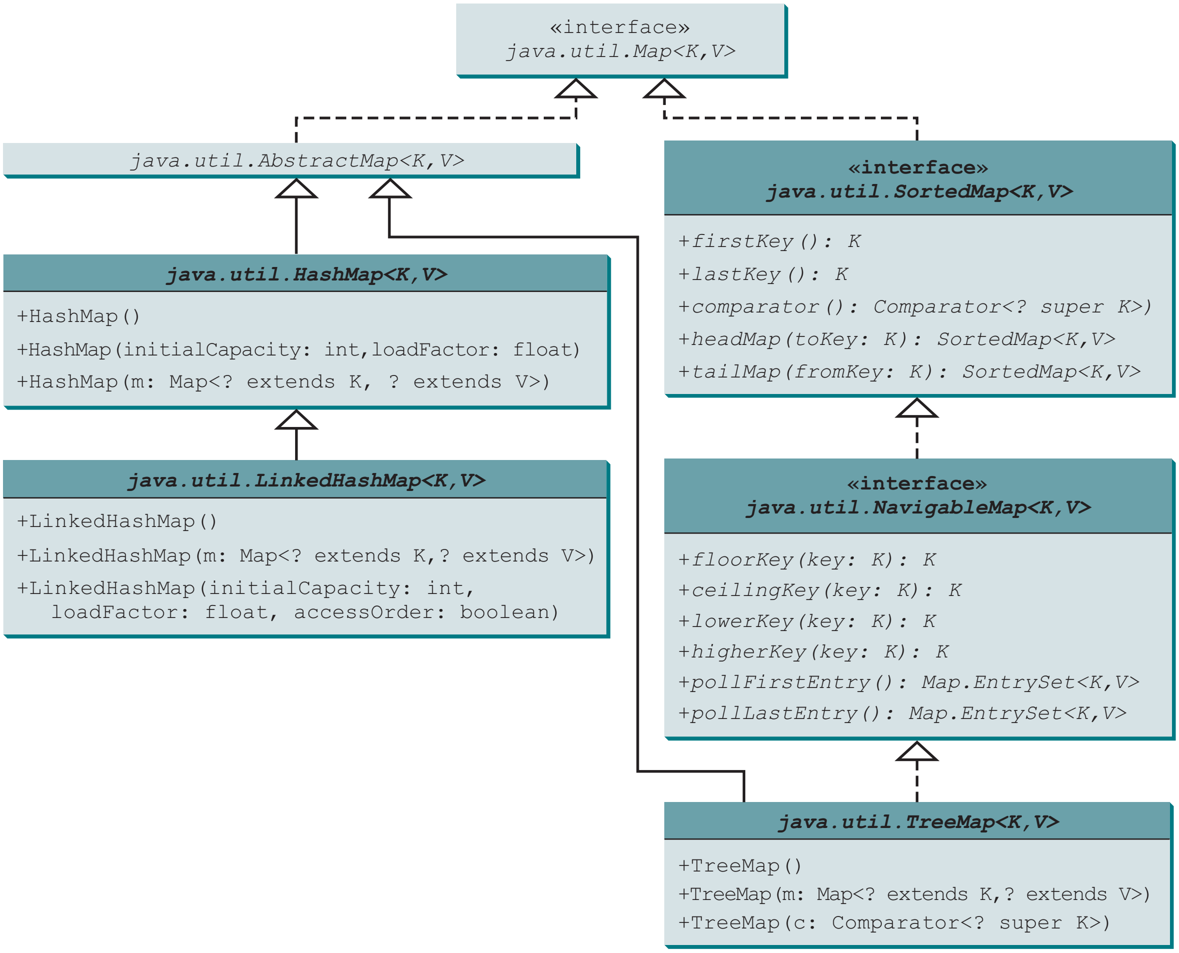

«interface»
java.util.Map<K,V>
java.util.AbstractMap<K,V>
java.util.HashMap<K,V>
+HashMap()
+HashMap(initialCapacity: int,loadFactor: float)
+HashMap(m: Map<? extends K, ? extends V>)
java.util.LinkedHashMap<K,V>
+LinkedHashMap()
+LinkedHashMap(m: Map<? extends K,? extends V>)
+LinkedHashMap(initialCapacity: int, loadFactor: float, accessOrder: boolean)
«interface»
java.util.SortedMap<K,V>
+firstKey(): K
+lastKey(): K
+comparator(): Comparator<? super K>)
+headMap(toKey: K): SortedMap<K,V>
+tailMap(fromKey: K): SortedMap<K,V>
«interface»
java.util.NavigableMap<K,V>
+floorKey(key: K): K
+ceilingKey(key: K): K
+lowerKey(key: K): K
+higherKey(key: K): K
+pollFirstEntry(): Map.EntrySet<K,V>
+pollLastEntry(): Map.EntrySet<K,V>
java.util.TreeMap<K,V>
+TreeMap()
+TreeMap(m: Map<? extends K,? extends V>)
+TreeMap(c: Comparator<? super K>)

FIGURE 21.6 The Java Collections Framework provides three concrete map classes.

The `HashMap` class is efficient for locating a value, inserting an entry, and deleting an entry.

`HashMap`

`LinkedHashMap` extends `HashMap` with a linked-list implementation that supports an ordering of the entries in the map. The entries in a `HashMap` are not ordered, but the entries in a `LinkedHashMap` can be retrieved either in the order in which they were inserted into the map (known as the *insertion order*) or in the order in which they were last accessed, from least recently to most recently accessed (*access order*). The no-arg constructor constructs a `LinkedHashMap` with the insertion order. To construct a `LinkedHashMap` with the access order, use `LinkedHashMap(initialCapacity, loadFactor, true)`.

`LinkedHashMap`
insertion order
access order

The `TreeMap` class is efficient for traversing the keys in a sorted order. The keys can be sorted using the `Comparable` interface or the `Comparator` interface. If you create a `TreeMap` using its no-arg constructor, the `compareTo` method in the `Comparable` interface is used to compare the keys in the map, assuming the class for the keys implements the `Comparable` interface. To use a comparator, you have to use the `TreeMap(Comparator comparator)` constructor to create a sorted map that uses the `compare` method in the comparator to order the entries in the map based on the keys.

`TreeMap`

`SortedMap` is a subinterface of `Map`, which guarantees the entries in the map are sorted. In addition, it provides the methods `firstKey()` and `lastKey()` for returning the first and the last keys in the map, and `headMap(toKey)` and `tailMap(fromKey)` for returning a portion of the map whose keys are less than `toKey` and greater than or equal to `fromKey`, respectively.

`SortedMap`

`NavigableMap` extends `SortedMap` to provide the navigation methods `lowerKey(key)`, `floorKey(key)`, `ceilingKey(key)`, and `higherKey(key)` that return keys, respectively, less than, less than or equal, greater than or equal, and greater than a given key and return `null` if there is no such key. The `pollFirstEntry()` and `pollLastEntry()` methods remove and return the first and the last entry in the tree map, respectively.

`NavigableMap`

Note

Prior to Java 2, `java.util.Hashtable` was used for mapping keys with values. `Hashtable` was redesigned to fit into the Java Collections Framework with all its methods retained for compatibility. `Hashtable` implements the `Map` interface and is used in the same way as `HashMap`, except that the update methods in `Hashtable` are synchronized.

`Hashtable`

Listing 21.8 gives an example that creates a *hash map*, a *linked hash map*, and a *tree map* for mapping students to ages. The program first creates a hash map with the student's name as its key and the age as its value. The program then creates a tree map from the hash map and displays the entries in ascending order of the keys. Finally, the program creates a linked hash map, adds the same entries to the map, and displays the entries.

hash map
linked hash map
tree map

LISTING 21.8 `TestMap.java`

```java
import java.util.*;

public class TestMap {
  public static void main(String[] args) {
    // Create a HashMap
    Map<String, Integer> hashMap = new HashMap<>();
    hashMap.put("Smith", 30);
    hashMap.put("Anderson", 31);
    hashMap.put("Lewis", 29);
    hashMap.put("Cook", 29);

    System.out.println("Display entries in HashMap");
    System.out.println(hashMap + "\n");

    // Create a TreeMap from the preceding HashMap
    Map<String, Integer> treeMap = new TreeMap<>(hashMap);
    System.out.println("Display entries in ascending order of key");
```

create map
add entry
tree map

linked hash map

forEach method

```
18     System.out.println(treeMap);
19
20     // Create a LinkedHashMap
21     Map<String, Integer> linkedHashMap =
22       new LinkedHashMap<>(16, 0.75f, true);
23     linkedHashMap.put("Smith", 30);
24     linkedHashMap.put("Anderson", 31);
25     linkedHashMap.put("Lewis", 29);
26     linkedHashMap.put("Cook", 29);
27
28     // Display the age for Lewis
29     System.out.println("\nThe age for " + "Lewis is " +
30       linkedHashMap.get("Lewis"));
31
32     System.out.println("Display entries in LinkedHashMap");
33     System.out.println(linkedHashMap);
34
35     // Display each entry with name and age
36     System.out.print("\nNames and ages are ");
37     treeMap.forEach(
38       (name, age) -> System.out.print(name + ": " + age + " "));
39   }
40 }
```

```
Display entries in HashMap
{Cook=29, Smith=30, Lewis=29, Anderson=31}

Display entries in ascending order of key
{Anderson=31, Cook=29, Lewis=29, Smith=30}

The age for Lewis is 29
Display entries in LinkedHashMap
{Smith=30, Anderson=31, Cook=29, Lewis=29}


Names and ages are Anderson: 31 Cook: 29 Lewis: 29 Smith: 30
```

As shown in the output, the entries in the **HashMap** are in random order. The entries in the **TreeMap** are in increasing order of the keys. The entries in the **LinkedHashMap** are in the order of their access, from least recently accessed to most recently.

All the concrete classes that implement the **Map** interface have at least two constructors. One is the no-arg constructor that constructs an empty map, and the other constructs a map from an instance of **Map**. Thus, **new TreeMap<>(hashMap)** (line 16) constructs a tree map from a hash map.

You can create an insertion- or access-ordered linked hash map. An access-ordered linked hash map is created in lines 21–22. The most recently accessed entry is placed at the end of the map. The entry with the key **Lewis** is last accessed in line 30, so it is displayed last in line 33.

It is convenient to process all the entries in the map using the **forEach** method. The program uses a **forEach** method to display a name and its age (lines 37–38).

Tip

If you don't need to maintain an order in a map when updating it, use a **HashMap**. When you need to maintain the insertion order or access order in the map, use a **LinkedHashMap**. When you need the map to be sorted on keys, use a **TreeMap**.

Check Point

21.5.1 How do you create an instance of **Map**? How do you add an entry to a map consisting of a key and a value? How do you remove an entry from a map? How do you find the size of a map? How do you traverse entries in a map?

21.5.2 Describe and compare `HashMap`, `LinkedHashMap`, and `TreeMap`.

21.5.3 Show the output of the following code:

```
import java.util.*;
public class Test {
  public static void main(String[] args) {
    Map<String, String> map = new LinkedHashMap<>();
    map.put("123", "John Smith");
    map.put("111", "George Smith");
    map.put("123", "Steve Yao");
    map.put("222", "Steve Yao");
    System.out.println("(1) " + map);
    System.out.println("(2) " + new TreeMap<String, String>(map));
    map.forEach((k, v) -> {
      if (k.equals("123")) System.out.println(v);});
  }
}
```

21.6 Case Study: Occurrences of Words

This case study writes a program that counts the occurrences of words in a text and displays the words and their occurrences in alphabetical order of the words.

Key Point

The program uses a `TreeMap` to store an entry consisting of a word and its count. For each word, check whether it is already a key in the map. If not, add an entry to the map with the word as the key and value `1`. Otherwise, increase the value for the word (key) by `1` in the map. Assume the words are case insensitive; for example, `Good` is treated the same as `good`.

Listing 21.9 gives the solution to the problem.

LISTING 21.9 CountOccurrenceOfWords.java

```
import java.util.*;

public class CountOccurrenceOfWords {
  public static void main(String[] args) {
    // Set text in a string
    String text = "Good morning. Have a good class. " +
      "Have a good visit. Have fun!";

    // Create a TreeMap to hold words as key and count as value
    Map<String, Integer> map = new TreeMap<>();

    String[] words = text.split("[\\s+\\p{P}]");
    for (int i = 0; i < words.length; i++) {
      String key = words[i].toLowerCase();

      if (key.length() > 0) {
        if (!map.containsKey(key)) {
          map.put(key, 1);
        }
        else {
          int value = map.get(key);
          value++;
          map.put(key, value);
        }
      }
    }

    // Display key and value for each entry
```

tree map

split string

add entry

update entry

display entry

```
29      map.forEach((k, v) -> System.out.println(k + "\t" + v));
30    }
31  }
```

```
a           2
class       1
fun         1
good        3
have        3
morning     1
visit       1
```

The program creates a **`TreeMap`** (line 10) to store pairs of words and their occurrence counts. The words serve as the keys. Since all values in the map must be stored as objects, the count is wrapped in an **`Integer`** object.

The program extracts a word from a text using the **`split`** method (line 12) in the **`String`** class (see Section 10.10.4 and Appendix H). The text is split into words using a whitespace **`\s`** or punctuation **`\p{P}`** as a delimiter. For each word extracted, the program checks whether it is already stored as a key in the map (line 17). If not, a new pair consisting of the word and its initial count (**`1`**) is stored in the map (line 18). Otherwise, the count for the word is incremented by **`1`** (lines 21–23).

The program displays the count and the key in each entry using the **`forEach`** method in the **`Map`** class (line 29).

Since the map is a tree map, the entries are displayed in increasing order of words. To display them in ascending order of the occurrence counts, see Programming Exercise 21.8.

Now sit back and think how you would write this program without using map. Your new program would be longer and more complex. You will find that map is a very efficient and powerful data structure for solving problems such as this.

Java Collections Framework provides comprehensive support of organizing and manipulating data. Suppose you wish to display the words in increasing order of their occurrence values, how do you modify the program? This can be done simply by creating a list of map entries and creating a **`Comparator`** for sorting the entries on their values as follows:

```
List<Map.Entry<String, Integer>> entries =
  new ArrayList<>(map.entrySet());
Collections.sort(entries, (entry1, entry2) -> {
  return entry1.getValue().compareTo(entry2.getValue()); });
for (Map.Entry<String, Integer> entry: entries) {
  System.out.println(entry.getKey() + "\t" + entry.getValue());
}
```

Check Point

21.6.1 Will the **`CountOccurrenceOfWords`** program work if line 10 is changed to

```
Map<String, int> map = new TreeMap<>();
```

21.6.2 Will the **`CountOccurrenceOfWords`** program work if line 17 is changed to

```
if (map.get(key) == null) {
```

21.6.3 Will the **`CountOccurrenceOfWords`** program work if line 29 is changed to

```
for (String key: map)
  System.out.println(key + "\t" + map.getValue(key));
```

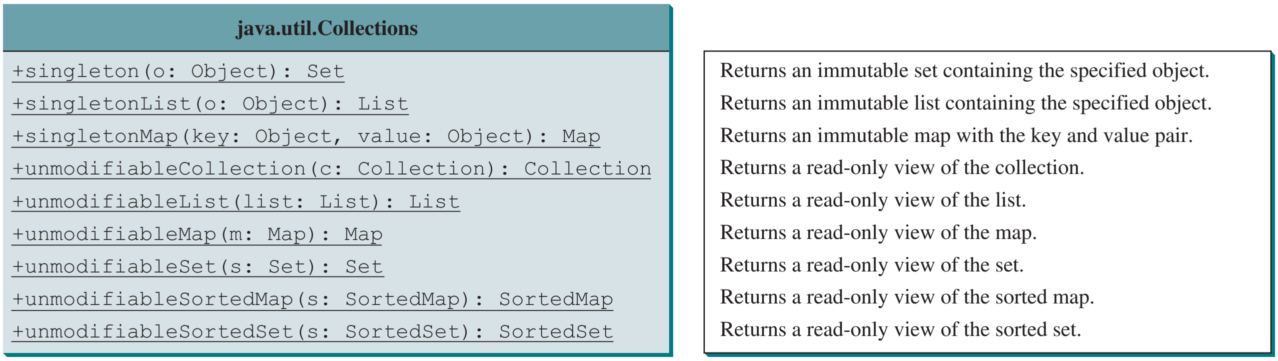
21.6.4 How do you simplify the code in lines 17–24 in Listing 21.9 in one line using a conditional expression?

21.7 Singleton and Unmodifiable Collections and Maps

You can create singleton sets, lists, and maps and unmodifiable sets, lists, and maps using the static methods in the `Collections` *class.*

Key Point

The `Collections` class contains the static methods for lists and collections. It also contains the methods for creating immutable singleton sets, lists, and maps and for creating read-only sets, lists, and maps, as shown in Figure 21.7.



java.util.Collections	
`+singleton(o: Object): Set`	Returns an immutable set containing the specified object.
`+singletonList(o: Object): List`	Returns an immutable list containing the specified object.
`+singletonMap(key: Object, value: Object): Map`	Returns an immutable map with the key and value pair.
`+unmodifiableCollection(c: Collection): Collection`	Returns a read-only view of the collection.
`+unmodifiableList(list: List): List`	Returns a read-only view of the list.
`+unmodifiableMap(m: Map): Map`	Returns a read-only view of the map.
`+unmodifiableSet(s: Set): Set`	Returns a read-only view of the set.
`+unmodifiableSortedMap(s: SortedMap): SortedMap`	Returns a read-only view of the sorted map.
`+unmodifiableSortedSet(s: SortedSet): SortedSet`	Returns a read-only view of the sorted set.

FIGURE 21.7 The `Collections` class contains the static methods for creating singleton and read-only sets, lists, and maps.

The `Collections` class defines three constants—`EMPTY_SET`, `EMPTY_LIST`, and `EMPTY_MAP`—for an empty set, an empty list, and an empty map. These collections are immutable. The class also provides the `singleton(Object o)` method for creating an immutable set containing only a single item, the `singletonList(Object o)` method for creating an immutable list containing only a single item, and the `singletonMap(Object key, Object value)` method for creating an immutable map containing only a single entry.

read-only view

The `Collections` class also provides six static methods for returning *read-only views for collections*: `unmodifiableCollection(Collection c)`, `unmodifiableList(List list)`, `unmodifiableMap(Map m)`, `unmodifiableSet(Set set)`, `unmodifiableSortedMap(SortedMap m)`, and `unmodifiableSortedSet(SortedSet s)`. This type of view is like a reference to the actual collection. However, you cannot modify the collection through a read-only view. Attempting to modify a collection through a read-only view will cause an `UnsupportedOperationException`.

In JDK 9, you can use the static `Set.of(e1, e2, ...)` method to create an immutable set and `Map.of(key1, value1, key2, value2, ...)` method to create an immutable map.

Check Point

21.7.1 What is wrong in the following code?

```
Set<String> set = Collections.singleton("Chicago");
set.add("Dallas");
```

21.7.2 What happens when you run the following code?

```
List list = Collections.unmodifiableList(Arrays.asList("Chicago",
  "Boston"));
list.remove("Dallas");
```

KEY TERMS

hash map	831
hash set	816
linked hash map	831
linked hash set	820
map	828
read-only view	835
set	816
tree map	831
tree set	821

CHAPTER SUMMARY

1. A set stores nonduplicate elements. To allow duplicate elements to be stored in a collection, you need to use a list.

2. A *map* stores key/value pairs. It provides a quick lookup for a value using a key.

3. Three types of sets are supported: `HashSet`, `LinkedHashSet`, and `TreeSet`. `HashSet` stores elements in an unpredictable order. `LinkedHashSet` stores elements in the order they were inserted. `TreeSet` stores elements sorted. `HashSet`, `LinkedHashSet`, and `TreeSet` are subtypes of `Collection`.

4. The `Map` interface maps keys to the elements. The keys are like indexes. In `List`, the indexes are integers. In `Map`, the keys can be any objects. A map cannot contain duplicate keys. Each key can map to at most one value. The `Map` interface provides the methods for querying, updating, and obtaining a collection of values and a set of keys.

5. Three types of maps are supported: `HashMap`, `LinkedHashMap`, and `TreeMap`. `HashMap` is efficient for locating a value, inserting an entry, and deleting an entry. `LinkedHashMap` supports ordering of the entries in the map. The entries in a `HashMap` are not ordered, but the entries in a `LinkedHashMap` can be retrieved either in the order in which they were inserted into the map (known as the *insertion order*) or in the order in which they were last accessed, from least recently accessed to most recently (*access order*). `TreeMap` is efficient for traversing the keys in a sorted order. The keys can be sorted using the `Comparable` interface or the `Comparator` interface.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

Sections 21.2–21.4

21.1 (*Perform set operations on hash sets*) Create two linked hash sets {`"George"`, `"Jim"`, `"John"`, `"Blake"`, `"Kevin"`, `"Michael"`} and {`"George"`, `"Katie"`, `"Kevin"`, `"Michelle"`, `"Ryan"`} and find their union, difference, and intersection. (You can clone the sets to preserve the original sets from being changed by these set methods.)

21.2 (*Display nondistinct words in ascending order*) Write a program that reads words from a text file and displays all the nonduplicate words in ascending order. The text file is passed as a command-line argument.

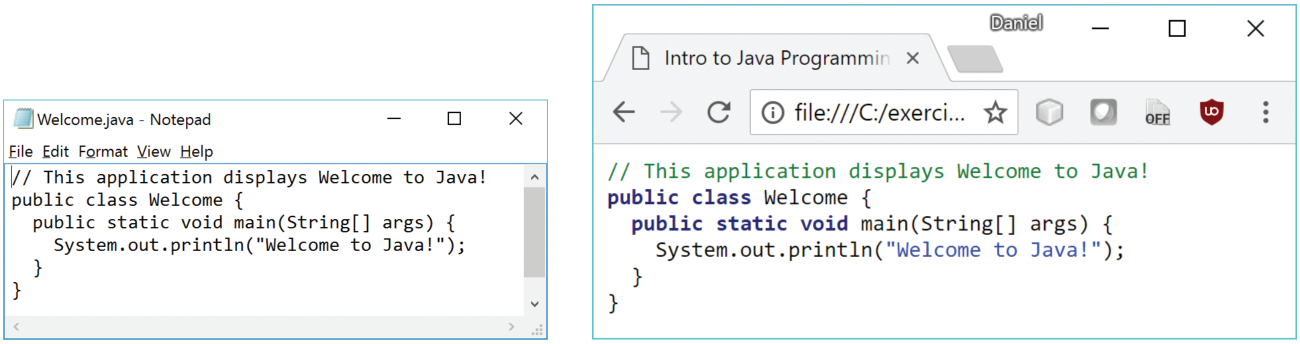
****21.3** (*Count the keywords in Java source code*) Revise the program in Listing 21.7. If a keyword is in a comment or in a string, don't count it. Pass the Java file name from the command line. Assume the Java source code is correct and line comments and paragraph comments do not overlap.

***21.4** (*Count consonants and vowels*) Write a program that prompts the user to enter a text file name and displays the number of vowels and consonants in the file. Use a set to store the vowels `A`, `E`, `I`, `O`, and `U`.

*****21.5** (*Syntax highlighting*) Write a program that converts a Java file into an HTML file. In the HTML file, the keywords, comments, and literals are displayed in bold navy, green, and blue, respectively. Use the command line to pass a Java file and an HTML file. For example, the following command

```
java Exercise21_05 Welcome.java Welcome.html
```

converts **Welcome.java** into **Welcome.html**. Figure 21.8a shows a Java file. The corresponding HTML file is shown in Figure 21.8b.



```
// This application displays Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

(a)

```
// This application displays Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

(b)

FIGURE 21.8 The Java code in plain text in (a) is displayed in HTML with syntax highlighted in (b). *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Sections 21.5–21.7

***21.6** (*Count the occurrences of numbers entered*) Write a program that reads an unspecified number of integers and finds the one that has the most occurrences. The input ends when the input is `0`. For example, if you entered `2 3 40 3 5 4 –3 3 3 2 0`, the number `3` occurred most often. If not one but several numbers have the most occurrences, all of them should be reported. For example, since `9` and `3` appear twice in the list `9 30 3 9 3 2 4`, both occurrences should be reported.

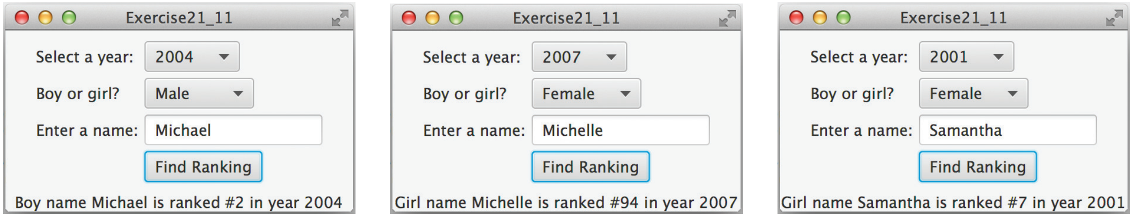
****21.7** (*Revise Listing 21.9, CountOccurrenceOfWords.java*) Rewrite Listing 21.9 to display the words in ascending order of occurrence counts.

****21.8** (*Count the occurrences of words in a text file*) Rewrite Listing 21.9 to read the text from a text file. The text file is passed as a command-line argument. Words are delimited by whitespace characters, punctuation marks (`,;.:?`), quotation marks (`'"`), and parentheses. Count words in case-insensitive fashion (e.g., consider `Good` and `good` to be the same word). The words must start with a letter. Display the output in alphabetical order of words, with each word preceded by its occurrence count.

****21.9** (*Guess the capitals using maps*) Rewrite Programming Exercise 8.37 to store pairs of each state and its capital in a map. Your program should prompt the user to enter a state and should display the capital for the state.

***21.10** (*Count the occurrences of each keyword*) Rewrite Listing 21.7, CountKeywords.java to read in a Java source-code file and count the occurrence of each keyword in the file, but don't count the keyword if it is in a comment or in a string literal.

****21.11** (*Baby name popularity ranking*) Use the data files from Programming Exercise 12.31 to write a program that enables the user to select a year, gender, and enter a name to display the ranking of the name for the selected year and gender, as shown in Figure 21.9. To achieve the best efficiency, create two arrays for boy's names and girl's names, respectively. Each array has 10 elements for 10 years. Each element is a map that stores a name and its ranking in a pair with the name as the key.



Select a year: 2004
Boy or girl? Male
Enter a name: Michael
Boy name Michael is ranked #2 in year 2004

Select a year: 2007
Boy or girl? Female
Enter a name: Michelle
Girl name Michelle is ranked #94 in year 2007

Select a year: 2001
Boy or girl? Female
Enter a name: Samantha
Girl name Samantha is ranked #7 in year 2001

Figure 21.9 The user selects a year and gender, enters a year, and clicks the Find Ranking button to display the ranking.
Source: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****21.12** (*Name for both genders*) Write a program that prompts the user to enter one of the filenames described in Programming Exercise 12.31 and displays the names that are used for both genders in the file. Use sets to store names and find common names in two sets. Here is a sample run:

```
Enter a file name for baby name ranking: babynamesranking2001.txt ↵Enter
69 names used for both genders
They are Tyler Ryan Christian ...
```

****21.13** (*Baby name popularity ranking*) Revise Programming Exercise 21.11 to prompt the user to enter year, gender, and name and display the ranking for the name. Prompt the user to enter another inquiry or exit the program. Here is a sample run:

```
Enter the year: 2010 ↵Enter
Enter the gender: M ↵Enter
Enter the name: Javier ↵Enter
Boy name Javier is ranked #190 in year 2010
Enter another inquiry? Y ↵Enter
Enter the year: 2001 ↵Enter
Enter the gender: F ↵Enter
Enter the name: Emily ↵Enter
Girl name Emily is ranked #1 in year 2001
Enter another inquiry? N ↵Enter
```

****21.14** (*Web crawler*) Rewrite Listing 12.18, WebCrawler.java, to improve the performance by using appropriate new data structures for `listOfPendingURLs` and `listofTraversedURLs`.

****21.15** (*Addition quiz*) Rewrite Programming Exercise 11.16 to store the answers in a set rather than a list.

CHAPTER 22

DEVELOPING EFFICIENT ALGORITHMS

Objectives

- To estimate algorithm efficiency using the Big *O* notation (§22.2).
- To explain growth rates and why constants and nondominating terms can be ignored in the estimation (§22.2).
- To determine the complexity of various types of algorithms (§22.3).
- To analyze the binary search algorithm (§22.4.1).
- To analyze the selection sort algorithm (§22.4.2).
- To analyze the Tower of Hanoi algorithm (§22.4.3).
- To describe common growth functions (constant, logarithmic, log-linear, quadratic, cubic, and exponential) (§22.4.4).
- To design efficient algorithms for finding Fibonacci numbers using dynamic programming (§22.5).
- To find the GCD using Euclid's algorithm (§22.6).
- To find prime numbers using the sieve of Eratosthenes (§22.7).
- To design efficient algorithms for finding the closest pair of points using the divide-and-conquer approach (§22.8).
- To solve the Eight Queens problem using the backtracking approach (§22.9).
- To design efficient algorithms for finding a convex hull for a set of points (§22.10).
- To design efficient algorithms for string matching using Boyer-Moore and KMP algorithms (§22.11).

22.1 Introduction

Key Point

Algorithm design is to develop a mathematical process for solving a problem. Algorithm analysis is to predict the performance of an algorithm.

The preceding two chapters introduced classic data structures (lists, stacks, queues, priority queues, sets, and maps) and applied them to solve problems. This chapter will use a variety of examples to introduce common algorithmic techniques (dynamic programming, divide-and-conquer, and backtracking) for developing efficient algorithms. Later in the book, we will introduce efficient algorithms in Chapters 23–29. Before introducing developing efficient algorithms, we need to address the question on how to measure algorithm efficiency.

22.2 Measuring Algorithm Efficiency Using Big O Notation

Key Point

The Big O notation obtains a function for measuring algorithm time complexity based on the input size. You can ignore multiplicative constants and nondominating terms in the function.

what is algorithm efficiency?

Suppose two algorithms perform the same task, such as search (linear search vs. binary search). Which one is better? To answer this question, you might implement these algorithms and run the programs to get execution times. However, there are two problems with this approach:

- First, many tasks run concurrently on a computer. The actual execution time of a particular program depends on the system load.
- Second, the actual execution time depends on specific input. Consider, for example, linear search and binary search. If an element to be searched happens to be the first in the list, linear search will find the element quicker than binary search.

growth rates

It is very difficult to compare algorithms by measuring their execution times. To overcome these problems, a theoretical approach was developed to analyze algorithm's running time independent of computers and specific input. This approach approximates the effect of a change on the size of the input. In this way, you can see how fast an algorithm's execution time increases as the input size increases, so you can compare two algorithms by examining their *growth rates*.

Big O notation

Consider linear search. The linear search algorithm compares the key with the elements in the array sequentially until the key is found or the array is exhausted. If the key is not in the array, it requires n comparisons for an array of size n. If the key is in the array, it requires $n/2$ comparisons on average. The algorithm's execution time is proportional to the size of the array. If you double the size of the array, you will expect the number of comparisons to double. The algorithm grows at a linear rate. The growth rate has an order of magnitude of n. Computer scientists use the Big O notation to represent the "order of magnitude." Using this notation, the complexity of the linear search algorithm is $O(n)$, pronounced as "*order of n*." We call an algorithm with a time complexity of $O(n)$ linear algorithm, and it exhibits a linear growth rate.

Note

The time complexity (a.k.a. running time) of an algorithm is the amount of the time taken by the algorithm measured using the Big O notation.

best-case input
worst-case input
average-case analysis

For the same input size, an algorithm's execution time may vary, depending on the input. An input that results in the shortest execution time is called the *best-case input*, and an input that results in the longest execution time is the *worst-case input.* Best-and worst-case analyses are to analyze the algorithms for their best- and worst-case inputs. Best- and worst-case analyses are not representative, but worst-case analysis is very useful. You can be assured that the algorithm will never be slower than the worst case. An *average-case analysis* attempts to determine the average amount of time among all possible inputs of the same size. Average-case analysis is ideal, but difficult to perform because for many problems it is hard to determine the relative probabilities and distributions of various input instances. Worst-case analysis is easier to perform, so the analysis is generally conducted for the worst case.

ignoring multiplicative constants

The linear search algorithm requires n comparisons in the worst case and $n/2$ comparisons in the average case if you are nearly always looking for something known to be in the list. Using the Big O notation, both the cases require $O(n)$ time. The multiplicative constant (1/2) can be omitted. Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates. The growth rate for $n/2$ or $100n$ is the same as for n, as illustrated in Table 22.1. Therefore, $O(n) = O(n/2) = O(100n)$.

TABLE 22.1 Growth Rates

n \ $f(n)$	n	$n/2$	$100n$	
100	100	50	10000	
200	200	100	20000	
	2	2	2	$f(200) / f(100)$

large input size

ignoring nondominating terms

Consider the algorithm for finding the maximum number in an array of n elements. To find the maximum number if n is 2, it takes one comparison and if n is 3, it takes two comparisons. In general, it takes $n - 1$ comparisons to find the maximum number in a list of n elements. Algorithm analysis is for large input size. If the input size is small, there is no significance in estimating an algorithm's efficiency. As n grows larger, the n part in the expression $n - 1$ dominates the complexity. The Big O notation allows you to ignore the nondominating part (e.g., -1 in the expression $n - 1$) and highlight the important part (e.g., n in the expression $n - 1$). Therefore, the complexity of this algorithm is $O(n)$.

constant time

The Big O notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take *constant time* with the notation $O(1)$. For example, a method that retrieves an element at a given index in an array takes constant time because the time does not grow as the size of the array increases.

useful summations

The following mathematical summations are often useful in algorithm analysis:

$$1 + 2 + 3 + \cdots + (n-2) + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

$$1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2} = O(n^2)$$

$$a^0 + a^1 + a^2 + a^3 + \cdots + a^{(n-1)} + a^n = \frac{a^{n+1} - 1}{a - 1} = O(a^n)$$

$$2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{(n-1)} + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1 = O(2^n)$$

Note

time complexity

space complexity

Time complexity is a measure of execution time using the Big O notation. Similarly, you can also measure *space complexity* using the Big O notation. *Space complexity* measures the amount of memory space used by an algorithm. The space complexity for most algorithms presented in this book is $O(n)$, that is, they exhibit linear growth rate to the input size. For example, the space complexity for linear search is $O(n)$.

Note

We presented the Big O notation in laymen's terms. Appendix J gives a precise mathematical definition for the Big O notation as well as the Big Omega and Big Theta notations.

Check Point

22.2.1 Why is a constant factor ignored in the Big O notation? Why is a nondominating term ignored in the Big O notation?

22.2.2 What is the order of each of the following functions?

$$\frac{(n^2 + 1)^2}{n}, \frac{(n^2 + \log^2 n^2)}{n}, n^3 + 100n^2 + n, 2^n + 100n^2 + 45n, n2^n + n^2 2^n$$

22.3 Examples: Determining Big *O*

Key Point

This section gives several examples of determining Big O for repetition, sequence, and selection statements.

Example 1

Consider the time complexity for the following loop:

```
for (int i = 1; i <= n; i++) {
  k = k + 5;
}
```

It is a constant time, c, for executing

```
k = k + 5;
```

Since the loop is executed n times, the time complexity for the loop is

$$T(n) = (\text{a constant } c) * n = O(n).$$

The theoretical analysis predicts the performance of the algorithm. To see how this algorithm performs, we run the code in Listing 22.1 to obtain the execution time for n = 1,000,000, 10,000,000, 100,000,000, and 1,000,000,000.

LISTING 22.1 PerformanceTest.java

input size 1,000,000
input size 10,000,000
input size 100,000,000
input size 1,000,000,000
time before execution
time after execution

```
public class PerformanceTest {
  public static void main(String[] args) {
    getTime(1000000);
    getTime(10000000);
    getTime(100000000);
    getTime(1000000000);
  }

  public static void getTime(long n) {
    long startTime = System.currentTimeMillis();
    long k = 0;
    for (long i = 1; i <= n; i++) {
      k = k + 5;
    }
    long endTime = System.currentTimeMillis();
    System.out.println("Execution time for n = " + n
      + " is " + (endTime - startTime) + " milliseconds");
  }
}
```

```
Execution time for n = 1,000,000 is 6 milliseconds
Execution time for n = 10,000,000 is 61 milliseconds
Execution time for n = 100,000,000 is 610 milliseconds
Execution time for n = 1,000,000,000 is 6048 milliseconds
```

Our analysis predicts a linear time complexity for this loop. As shown in the sample output, when the input size increases 10 times, the runtime increases roughly 10 times. The execution confirms to the prediction.

Example 2

What is the time complexity for the following loop?

```
for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= n; j++) {
    k = k + i + j;
  }
}
```

It is a constant time, c, for executing

```
k = k + i + j;
```

The outer loop executes n times. For each iteration in the outer loop, the inner loop is executed n times. Thus, the time complexity for the loop is

$$T(n) = (\text{a constant } c) * n * n = O(n^2)$$

quadratic time

An algorithm with the $O(n^2)$ time complexity is called a *quadratic algorithm* and it exhibits a quadratic growth rate. The quadratic algorithm grows quickly as the problem size increases. If you double the input size, the time for the algorithm is quadrupled. Algorithms with a nested loop are often quadratic.

Example 3

Consider the following loop:

```
for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= i; j++) {
    k = k + i + j;
  }
}
```

The outer loop executes n times. For $i = 1, 2, \ldots,$ the inner loop is executed one time, two times, and n times. Thus, the time complexity for the loop is

$$\begin{aligned} T(n) &= c + 2c + 3c + 4c + \cdots + nc \\ &= cn(n + 1)/2 \\ &= (c/2)\, n^2 + (c/2)n \\ &= O(n^2) \end{aligned}$$

Example 4

Consider the following loop:

```
for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= 20; j++) {
    k = k + i + j;
  }
}
```

The inner loop executes 20 times and the outer loop n times. Therefore, the time complexity for the loop is

$$T(n) = 20 * c * n = O(n)$$

Example 5

Consider the following sequences:

```
for (int j = 1; j <= 10; j++) {
  k = k + 4;
}
```

```
for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= 20; j++) {
    k = k + i + j;
  }
}
```

The first loop executes 10 times and the second loop 20 * n times. Thus, the time complexity for the loop is

$$T(n) = 10 * c + 20 * c * n = O(n)$$

Example 6

Consider the following selection statement:

```
if (list.contains(e)) {
  System.out.println(e);
}
else
  for (Object t: list) {
    System.out.println(t);
  }
```

Suppose the list contains n elements. The execution time for `list.contains(e)` is $O(n)$. The loop in the `else` clause takes $O(n)$ time. Hence, the time complexity for the entire statement is

$$\begin{aligned} T(n) &= \text{if test time} + \text{worst-case time (if clause, else clause)} \\ &= O(n) + O(n) = O(n) \end{aligned}$$

Example 7

Consider the computation of a^n for an integer n. A simple algorithm would multiply a n times, as follows:

```
result = 1;
for (int i = 1; i <= n; i++)
  result *= a;
```

The algorithm takes $O(n)$ time. Without loss of generality, assume that $n = 2^k$. You can improve the algorithm using the following scheme:

```
result = a;
for (int i = 1; i <= k; i++)
  result = result * result;
```

The algorithm takes $O(\log n)$ time. For an arbitrary n, you can revise the algorithm and prove that the complexity is still $O(\log n)$. (See CheckPoint Question 22.3.5.)

Note

An algorithm with the $O(\log n)$ time complexity is called a *logarithmic algorithm* and it exhibits a logarithmic growth rate. The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted. In algorithm analysis, the base is usually 2.

omitting base

Check Point

22.3.1 Count the number of iterations in the following loops.

```
int count = 1;
while (count < 30) {
  count = count * 2;
}
```

(a)

```
int count = 15;
while (count < 30) {
  count = count * 3;
}
```

(b)

```
int count = 1;
while (count < n) {
  count = count *  2;
}
```

(c)

```
int count = 15;
while (count < n) {
  count = count *  3;
}
```

(d)

22.3.2 How many stars are displayed in the following code if `n` is 10? How many if `n` is 20? Use the Big *O* notation to estimate the time complexity.

```
for (int i = 0; i < n; i++) {
  System.out.print('*');
}
```

(a)

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    System.out.print('*');
  }
}
```

(b)

```
for (int k = 0; k < n; k++) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      System.out.print('*');
    }
  }
}
```

(c)

```
for (int k = 0; k < 10; k++) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      System.out.print('*');
    }
  }
}
```

(d)

22.3.3 Use the Big *O* notation to estimate the time complexity of the following methods:

```
public static void mA(int n) {
  for (int i = 0; i < n; i++) {
    System.out.print(Math.random());
  }
}
```

(a)

```
public static void mB(int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++)
      System.out.print(Math.random());
  }
}
```

(b)

```
public static void mC(int[ ] m) {
  for (int i = 0; i < m.length; i++) {
    System.out.print(m[i]);
  }

  for (int i = m.length – 1; i >= 0; )
  {
    System.out.print(m[i]);
    i--;
  }
}
```

(c)

```
public static void mD(int[] m) {
  for (int i = 0; i < m.length; i++) {
    for (int j = 0; j < i; j++)
      System.out.print(m[i] * m[j]);
  }
}
```

(d)

22.3.4 Design an $O(n)$ time algorithm for computing the sum of numbers from $n1$ to $n2$ for ($n1 < n2$). Can you design an $O(1)$ for performing the same task?

22.3.5 Example 7 in Section 22.3 assumes $n = 2^k$. Revise the algorithm for an arbitrary n and prove that the complexity is still $O(\log n)$.

22.4 Analyzing Algorithm Time Complexity

Key Point

This section analyzes the complexity of several well-known algorithms: binary search, selection sort, and Tower of Hanoi.

22.4.1 Analyzing Binary Search

binary search animation on the Companion Website

The binary search algorithm presented in Listing 7.7, BinarySearch.java, searches for a key in a sorted array. Each iteration in the algorithm contains a fixed number of operations, denoted by c. Let $T(n)$ denote the time complexity for a binary search on a list of n elements. Without loss of generality, assume n is a power of 2 and $k = \log n$. Since a binary search eliminates half of the input after two comparisons,

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{2^2}\right) + c + c = T\left(\frac{n}{2^k}\right) + kc \\ &= T(1) + c\log n = 1 + (\log n)c \\ &= O(\log n) \end{aligned}$$

logarithmic time

Ignoring constants and nondominating terms, the complexity of the binary search algorithm is $O(\log n)$. This is a logarithmic algorithm. The logarithmic algorithm grows slowly as the problem size increases. In the case of binary search, each time you double the array size, at most one more comparison will be required. If you square the input size of any logarithmic-time algorithm, you only double the time of execution. Therefore, a logarithmic-time algorithm is very efficient.

22.4.2 Analyzing Selection Sort

selection sort animation on the Companion Website

The selection sort algorithm presented in Listing 7.8, SelectionSort.java, finds the smallest element in the list and swaps it with the first element. It then finds the smallest element remaining and swaps it with the first element in the remaining list, and so on until the remaining list contains only one element left to be sorted. The number of comparisons is $n - 1$ for the first iteration, $n - 2$ for the second iteration, and so on. Let $T(n)$ denote the complexity for selection sort and c denote the total number of other operations such as assignments and additional comparisons in each iteration. Thus,

$$\begin{aligned} T(n) &= (n - 1) + c + (n - 2) + c + \cdots + 2 + c + 1 + c \\ &= \frac{(n-1)(n-1+1)}{2} + c(n-1) = \frac{n^2}{2} - \frac{n}{2} + cn - c \\ &= O(n^2) \end{aligned}$$

Therefore, the complexity of the selection sort algorithm is $O(n^2)$.

22.4.3 Analyzing the Tower of Hanoi Problem

The Tower of Hanoi problem presented in Listing 18.8, TowerOfHanoi.java, recursively moves n disks from tower A to tower B with the assistance of tower C as follows:

1. Move the first $n - 1$ disks from A to C with the assistance of tower B.

2. Move disk n from A to B.
3. Move $n - 1$ disks from C to B with the assistance of tower A.

The complexity of this algorithm is measured by the number of moves. Let $T(n)$ denote the number of moves for the algorithm to move n disks from tower A to tower B with $T(1) = 1$. Thus,

$$\begin{aligned} T(n) &= T(n-1) + 1 + T(n-1) \\ &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 2(2(2T(n-3) + 1) + 1) + 1 \\ &= 2^{n-1}T(1) + 2^{n-2} + \cdots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 = (2^n - 1) = O(2^n) \end{aligned}$$

An algorithm with $O(2^n)$ time complexity is called an *exponential algorithm* and it exhibits an exponential growth rate. As the input size increases, the time for the exponential algorithm grows exponentially. Exponential algorithms are not practical for large input size. Suppose the disk is moved at a rate of 1 per second. It would take $2^{32}/(365 * 24 * 60 * 60) = 136$ years to move 32 disks and $2^{64}/(365 * 24 * 60 * 60) = 585$ billion years to move 64 disks.

$O(2^n)$
exponential time

22.4.4 Common Recurrence Relations

Recurrence relations are a useful tool for analyzing algorithm complexity. As shown in the preceding examples, the complexity for binary search, selection sort, and the Tower of Hanoi is $T(n) = T\left(\frac{n}{2}\right) + c$, $T(n) = T(n-1) + O(n)$, and $T(n) = 2T(n-1) + O(1)$, respectively. Table 22.2 summarizes the common recurrence relations.

TABLE 22.2 Common Recurrence Functions

Recurrence Relation	*Result*	*Example*
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	CheckPoint Question 22.8.3
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Merge sort (Chapter 23)
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Tower of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

22.4.5 Comparing Common Growth Functions

The preceding sections analyzed the complexity of several algorithms. Table 22.3 lists some common growth functions and shows how growth rates change as the input size doubles from $n = 25$ to $n = 50$.

TABLE 22.3 Change of Growth Rates

Function	*Name*	$n = 25$	$n = 50$	$f(50)$ and $f(25)$
$O(1)$	Constant time	1	1	1
$O(\log n)$	Logarithmic time	4.64	5.64	1.21
$O(n)$	Linear time	25	50	2
$O(n \log n)$	Log-linear time	116	282	2.43
$O(n^2)$	Quadratic time	625	2,500	4
$O(n^3)$	Cubic time	15,625	125,000	8
$O(2^n)$	Exponential time	3.36×10^7	1.27×10^{15}	3.35×10^7

These functions are ordered as follows, as illustrated in Figure 22.1.

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

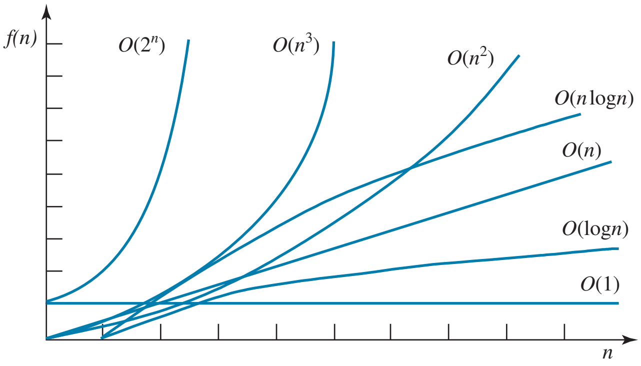


FIGURE 22.1 As the size n increases, the function grows.

Check Point

22.4.1 Put the following growth functions in order:

$$\frac{5n^3}{4{,}032},\ 44 \log n,\ 10n \log n,\ 500,\ 2n^2,\ \frac{2^n}{45},\ 3n$$

22.4.2 Estimate the time complexity for adding two $n \times m$ matrices and multiplying an $n \times m$ matrix by an $m \times k$ matrix.

22.4.3 Describe an algorithm for finding the occurrence of the max element in an array. Analyze the complexity of the algorithm.

22.4.4 Describe an algorithm for removing duplicates from an array. Analyze the complexity of the algorithm.

22.4.5 Analyze the following sorting algorithm:

```
for (int i = 0; i < list.length - 1; i++) {
  if (list[i] > list[i + 1]) {
    swap list[i] with list[i + 1];
    i = -1;
  }
}
```

22.4.6 Analyze the complexity for computing a polynomial $f(x)$ of degree n for a given x value using a brute-force approach and the Horner's approach, respectively. A brute-force approach is to compute each term in the polynomial and add them together. The Horner's approach was introduced in Section 6.7.

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x^1 + a_0$$

22.5 Finding Fibonacci Numbers Using Dynamic Programming

This section analyzes and designs an efficient algorithm for finding Fibonacci numbers using dynamic programming.

Key Point

Section 18.3, Case Study: Computing Fibonacci Numbers, gave a recursive method for finding the Fibonacci number, as follows:

```java
/** The method for finding the Fibonacci number */
public static long fib(long index) {
  if (index == 0) // Base case
    return 0;
  else if (index == 1) // Base case
    return 1;
  else // Reduction and recursive calls
    return fib(index - 1) + fib(index - 2);
}
```

We can now prove that the complexity of this algorithm is $O(2^n)$. For convenience, let the index be n. Let $T(n)$ denote the complexity for the algorithm that finds fib(n), and c denote the constant time for comparing the index with **`0`** and **`1`**. Thus,

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c \\ &\le 2T(n-1) + c \\ &\le 2(2T(n-2) + c) + c \\ &= 2^2T(n-2) + 2c + c \end{aligned}$$

Similar to the analysis of the Tower of Hanoi problem, we can show that $T(n)$ is $O(2^n)$.

This algorithm is not efficient. Is there an efficient algorithm for finding a Fibonacci number? The trouble with the recursive **`fib`** method is that the method is invoked redundantly with the same arguments. For example, to compute **`fib(4)`**, **`fib(3)`** and **`fib(2)`** are invoked. To compute **`fib(3)`**, **`fib(2)`** and **`fib(1)`** are invoked. Note **`fib(2)`** is redundantly invoked. We can improve it by avoiding repeatedly calling of the **`fib`** method with the same argument. Note a new Fibonacci number is obtained by adding the preceding two numbers in the sequence. If you use the two variables **`f0`** and **`f1`** to store the two preceding numbers, the new number, **`f2`**, can be immediately obtained by adding **`f0`** with **`f1`**. Now you should update **`f0`** and **`f1`** by assigning **`f1`** to **`f0`** and assigning **`f2`** to **`f1`**, as shown in Figure 22.2.

```
                    f0 f1 f2
Fibonacci  series : 0  1  1  2  3  5  8  13  21  34  55  89 . . .
           indices : 0  1  2  3  4  5  6  7   8   9   10  11
                       f0 f1 f2
Fibonacci  series : 0  1  1  2  3  5  8  13  21  34  55  89 . . .
           indices : 0  1  2  3  4  5  6  7   8   9   10  11
                                                  f0  f1  f2
Fibonacci  series : 0  1  1  2  3  5  8  13  21  34  55  89 . . .
           indices : 0  1  2  3  4  5  6  7   8   9   10  11
```

Figure 22.2 Variables **`f0`**, **`f1`**, and **`f2`** store three consecutive Fibonacci numbers in the series.

The new method is implemented in Listing 22.2.

Listing 22.2 `ImprovedFibonacci.java`

input

invoke `fib`

`f0`

`f1`

`f2`

update `f0`, `f1`, `f2`

```java
import java.util.Scanner;

public class ImprovedFibonacci {
  /** Main method */
  public static void main(String args[]) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an index for the Fibonacci number: ");
    int index = input.nextInt();

    // Find and display the Fibonacci number
    System.out.println(
      "Fibonacci number at index " + index + " is " + fib(index));
  }

  /** The method for finding the Fibonacci number */
  public static long fib(long n) {
    long f0 = 0; // For fib(0)
    long f1 = 1; // For fib(1)
    long f2 = 1; // For fib(2)

    if (n == 0)
      return f0;
    else if (n == 1)
      return f1;
    else if (n == 2)
      return f2;

    for (int i = 3; i <= n; i++) {
      f0 = f1;
      f1 = f2;
      f2 = f0 + f1;
    }

    return f2;
  }
}
```

```
Enter an index for the Fibonacci number: 6 ↵Enter
Fibonacci number at index 6 is 8
```

```
Enter an index for the Fibonacci number: 7 ↵Enter
Fibonacci number at index 7 is 13
```

$O(n)$

Obviously, the complexity of this new algorithm is $O(n)$. This is a tremendous improvement over the recursive $O(2^n)$ algorithm.

dynamic programming

Algorithm Design Note

The algorithm for computing Fibonacci numbers presented here uses an approach known as *dynamic programming*. Dynamic programming is the process of solving subproblems, and then combining the solutions of the subproblems to obtain an overall solution. This naturally leads to a recursive solution. However, it would be inefficient to use recursion because the subproblems overlap. The key idea behind dynamic programming is to solve each subproblem only once and store the results for subproblems for later use to avoid redundant computing of the subproblems.

Check Point

22.5.1 What is dynamic programming? Give an example of dynamic programming.

22.5.2 Why is the recursive Fibonacci algorithm inefficient, but the nonrecursive Fibonacci algorithm efficient?

22.6 Finding Greatest Common Divisors Using Euclid's Algorithm

Key Point

This section presents several algorithms in the search for an efficient algorithm for finding the greatest common divisor of two integers.

GCD

brute force

The greatest common divisor (GCD) of two integers is the largest number that divides both integers. Listing 5.9, GreatestCommonDivisor.java, presented a brute-force algorithm for finding the GCD of two integers `m` and `n`.

Algorithm Design Note

Brute force refers to an algorithmic approach that solves a problem in the simplest or most direct or obvious way. As a result, such an algorithm can end up doing far more work to solve a given problem than a cleverer or more sophisticated algorithm might do. On the other hand, a brute-force algorithm is often easier to implement than a more sophisticated one.

The brute-force algorithm checks whether `k` (for `k` = `2`, `3`, `4`, and so on) is a common divisor for `n1` and `n2`, until `k` is greater than `n1` or `n2`. The algorithm can be described as follows:

```
public static int gcd(int m, int n) {
  int gcd = 1;

  for (int k = 2; k <= m && k <= n; k++) {
    if (m % k == 0 && n % k == 0)
      gcd = k;
  }

  return gcd;
}
```

assume $m \geq n$

$O(n)$

improved solutions

Assuming $m \geq n$, the complexity of this algorithm is obviously $O(n)$.

Is there a better algorithm for finding the GCD? Rather than searching a possible divisor from `1` up, it is more efficient to search from `n` down. Once a divisor is found, the divisor is the GCD. Therefore, you can improve the algorithm using the following loop:

```
for (int k = n; k >= 1; k--) {
  if (m % k == 0 && n % k == 0) {
    gcd = k;
    break;
  }
}
```

This algorithm is better than the preceding one, but its worst-case time complexity is still $O(n)$.

A divisor for a number `n` cannot be greater than `n / 2`, so you can further improve the algorithm using the following loop:

```
for (int k = m / 2; k >= 1; k--) {
  if (m % k == 0 && n % k == 0) {
    gcd = k;
    break;
  }
}
```

However, this algorithm is incorrect because `n` can be a divisor for `m`. This case must be considered. The correct algorithm is shown in Listing 22.3.

LISTING 22.3 `GCD.java`

check divisor

GCD found

input

input

```java
import java.util.Scanner;

public class GCD {
  /** Find GCD for integers m and n */
  public static int gcd(int m, int n) {
    int gcd = 1;

    if (m % n == 0)  return n;

    for (int k = n /  2; k >= 1; k--) {
      if (m % k == 0 && n % k == 0) {
        gcd = k;
        break;
      }
    }

    return gcd;
  }

  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter first integer: ");
    int m = input.nextInt();
    System.out.print("Enter second integer: ");
    int n = input.nextInt();

    System.out.println("The greatest common divisor for " + m +
      " and " + n + " is " + gcd(m, n));
  }
}
```

```
Enter first integer: 2525 ↵Enter
Enter second integer: 125 ↵Enter
The greatest common divisor for 2525 and 125 is 25
```

```
Enter first integer: 3 ↵Enter
Enter second integer: 3 ↵Enter
The greatest common divisor for 3 and 3 is 3
```

Assuming $m \geq n$, the `for` loop is executed at most $n/2$ times, which cuts the time by half from the previous algorithm. The time complexity of this algorithm is still $O(n)$, but practically, it is much faster than the algorithm in Listing 5.9.

O(n)

Note

The Big *O* notation provides a good theoretical estimate of algorithm efficiency. However, two algorithms of the same time complexity are not necessarily equally efficient. As shown in the preceding example, both algorithms in Listing 5.9 and Listing 22.3 have the same complexity, but in practice, the one in Listing 22.3 is obviously better.

practical consideration

Euclid's algorithm

A more efficient algorithm for finding the GCD was discovered by Euclid around 300 B.C. This is one of the oldest known algorithms. It can be defined recursively as follows:

Let `gcd(m, n)` denote the GCD for integers `m` and `n`:

- If `m % n` is `0`, `gcd(m, n)` is `n`.
- Otherwise, `gcd(m, n)` is `gcd(n, m % n)`.

It is not difficult to prove the correctness of this algorithm. Suppose `m % n = r`. Thus, `m = qn + r`, where `q` is the quotient of `m / n`. Any number that divides `m` and `n` evenly must also divide `r` evenly. Therefore, `gcd(m, n)` is the same as `gcd(n, r)`, where `r = m % n`. The algorithm can be implemented as in Listing 22.4.

LISTING 22.4 GCDEuclid.java

```java
import java.util.Scanner;

public class GCDEuclid {
  /** Find GCD for integers m and n */
  public static int gcd(int m, int n) {
    if (m % n == 0)
      return n;
    else
      return gcd(n, m % n);
  }

  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter first integer: ");
    int m = input.nextInt();
    System.out.print("Enter second integer: ");
    int n = input.nextInt();

    System.out.println("The greatest common divisor for " + m +
      " and " + n + " is " + gcd(m, n));
  }
}
```

base case

reduction

input

input

```
Enter first integer: 2525 ↵Enter
Enter second integer: 125 ↵Enter
The greatest common divisor for 2525 and 125 is 25
```

```
Enter first integer: 3 ↵Enter
Enter second integer: 3 ↵Enter
The greatest common divisor for 3 and 3 is 3
```

best case

average case

worst case

In the best case when `m % n` is `0`, the algorithm takes just one step to find the GCD. It is difficult to analyze the average case. However, we can prove that the worst-case time complexity is $O(\log n)$.

Assuming $m \geq n$, we can show that `m % n < m / 2`, as follows:

- If `n <= m / 2`, `m % n < m / 2`, since the remainder of m divided by n is always less than n.
- If `n > m / 2`, `m % n = m – n < m / 2`. Therefore, `m % n < m / 2`.

Euclid's algorithm recursively invokes the `gcd` method. It first calls `gcd(m, n)`, then calls `gcd(n, m % n)` and `gcd(m % n, n % (m % n))`, and so on, as follows:

```
  gcd(m, n)
= gcd(n, m % n)
= gcd(m % n, n % (m % n))
= . . .
```

Since `m % n < m / 2` and `n % (m % n) < n / 2`, the argument passed to the `gcd` method is reduced by half after every two iterations. After invoking `gcd` two times, the second parameter is less than $n/2$. After invoking `gcd` four times, the second parameter is less than $n/4$. After invoking `gcd` six times, the second parameter is less than $\frac{n}{2^3}$. Let k be the number of times the `gcd` method is invoked. After invoking `gcd` k times, the second parameter is less than $\frac{n}{2^{(k/2)}}$, which is greater than or equal to 1. That is,

$$\frac{n}{2^{(k/2)}} \geq 1 \quad \Rightarrow \quad n \geq 2^{(k/2)} \quad \Rightarrow \quad \log n \geq k/2 \quad \Rightarrow \quad k \leq 2 \log n$$

Therefore, $k \leq 2 \log n$. Thus, the time complexity of the `gcd` method is $O(\log n)$.

The worst case occurs when the two numbers result in the most divisions. It turns out that two successive Fibonacci numbers will result in the most divisions. Recall that the Fibonacci series begins with 0 and 1, and each subsequent number is the sum of the preceding two numbers in the series, such as:

0 1 1 2 3 5 8 13 21 34 55 89 . . .

The series can be recursively defined as

```
fib(0) = 0;
fib(1) = 1;
fib(index) = fib(index – 2) + fib(index – 1); index >= 2
```

For two successive Fibonacci numbers `fib(index)` and `fib(index – 1)`,

```
gcd(fib(index), fib(index – 1))
= gcd(fib(index – 1), fib(index – 2))
= gcd(fib(index – 2), fib(index – 3))
= gcd(fib(index – 3), fib(index – 4))
= . . .
= gcd(fib(2), fib(1))
= 1
```

For example,

```
gcd(21, 13)
= gcd(13, 8)
= gcd(8, 5)
= gcd(5, 3)
= gcd(3, 2)
= gcd(2, 1)
= 1
```

Therefore, the number of times the `gcd` method is invoked is the same as the index. We can prove that $index \leq 1.44 \log n$, where $n = \text{fib}(\text{index} - 1)$. This is a tighter bound than $\text{index} \leq 2 \log n$.

Table 22.4 summarizes the complexity of three algorithms for finding the GCD.

TABLE 22.4 Comparisons of GCD Algorithms

Algorithm	*Complexity*	*Description*
Listing 5.9	$O(n)$	Brute-force, checking all possible divisors
Listing 22.3	$O(n)$	Checking half of all possible divisors
Listing 22.4	$O(\log n)$	Euclid's algorithm

Check Point

22.6.1 Prove the following algorithm for finding the GCD of the two integers `m` and `n` is incorrect:

```
int gcd = 1;
for (int k = Math.min(Math.sqrt(n), Math.sqrt(m)); k >= 1; k--) {
  if (m % k == 0 && n % k == 0) {
    gcd = k;
    break;
  }
}
```

22.7 Efficient Algorithms for Finding Prime Numbers

Key Point

This section presents several algorithms in the search for an efficient algorithm for finding prime numbers.

A $150,000 award awaits the first individual or group who discovers a prime number with at least 100,000,000 decimal digits (https://www.eff.org/awards/coop).

Can you design a fast algorithm for finding prime numbers?

what is prime?

An integer greater than `1` is *prime* if its only positive divisor is `1` or itself. For example, `2`, `3`, `5`, and `7` are prime numbers, but `4`, `6`, `8`, and `9` are not.

How do you determine whether a number `n` is prime? Listing 5.15 presented a brute-force algorithm for finding prime numbers. The algorithm checks whether `2`, `3`, `4`, `5`, . . . , or `n – 1` divides `n`. If not, `n` is prime. This algorithm takes $O(n)$ time to check whether `n` is prime. Note you need to check only whether `2`, `3`, `4`, `5`, . . . , or `n/2` divides `n`. If not, `n` is prime. This algorithm is slightly improved, but it is still of $O(n)$.

In fact, we can prove that if `n` is not a prime, `n` must have a factor that is greater than `1` and less than or equal to $\sqrt{n}$. Here is the proof. Since `n` is not a prime, there exist two numbers `p` and `q` such that `n` = `pq` with $1 < p \leq q$. Note that $n = \sqrt{n}\sqrt{n}$. `p` must be less than or equal to $\sqrt{n}$. Hence, you need to check only whether `2`, `3`, `4`, `5`, . . . , or $\sqrt{n}$ divides `n`. If not, `n` is prime. This significantly reduces the time complexity of the algorithm to $O(\sqrt{n})$.

Now consider the algorithm for finding all the prime numbers up to `n`. A straightforward implementation is to check whether `i` is prime for $i =$ `2`, `3`, `4`, . . . , `n`. The program is given in Listing 22.5.

Listing 22.5 PrimeNumbers.java

check prime

increase count

check next number

```java
import java.util.Scanner;

public class PrimeNumbers {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Find all prime numbers <= n, enter n: ");
    int n = input.nextInt();

    final int NUMBER_PER_LINE = 10; // Display 10 per line
    int count = 0; // Count the number of prime numbers
    int number = 2; // A number to be tested for primeness

    System.out.println("The prime numbers are:");

    // Repeatedly find prime numbers
    while (number <= n) {
      // Assume the number is prime
      boolean isPrime = true; // Is the current number prime?

      // Test if number is prime
      for (int divisor = 2; divisor <= (int)(Math.sqrt(number));
          divisor++) {
        if (number % divisor == 0) {  // If true, number is not prime
          isPrime = false; // Set isPrime to false
          break; // Exit the for loop
        }
      }

      // Print the prime number and increase the count
      if (isPrime) {
        count++; // Increase the count

        if (count % NUMBER_PER_LINE == 0) {
          // Print the number and advance to the new line
          System.out.printf("%7d\n", number);
        }
        else
          System.out.printf("%7d", number);
      }

      // Check if the next number is prime
      number++;
    }

    System.out.println("\n" + count +
      " prime(s) less than or equal to " + n);
  }
}
```

```
Find all prime numbers <= n, enter n: 1000 ↵Enter
The prime numbers are:
      2      3      5      7     11     13     17     19     23     29
     31     37     41     43     47     53     59     61     67     71
...
...
168 prime(s) less than or equal to 1000
```

The program is not efficient if you have to compute `Math.sqrt(number)` for every iteration of the `for` loop (line 21). A good compiler should evaluate `Math.sqrt(number)` only once for the entire `for` loop. To ensure this happens, you can explicitly replace line 21 with the following two lines:

```
int squareRoot = (int)(Math.sqrt(number));
for (int divisor = 2; divisor <= squareRoot; divisor++) {
```

In fact, there is no need to actually compute `Math.sqrt(number)` for every `number`. You need to look only for the perfect squares such as `4`, `9`, `16`, `25`, `36`, `49`, and so on. Note for all the numbers between `36` and `48`, inclusively, their `(int)(Math.sqrt(number))` is `6`. With this insight, you can replace the code in lines 16–26 with the following:

```
...
int squareRoot = 1;

// Repeatedly find prime numbers
while (number <= n) {
  // Assume the number is prime
  boolean isPrime = true; // Is the current number prime?

  if (squareRoot * squareRoot < number) squareRoot++;

  // Test if number is prime
  for (int divisor = 2; divisor <= squareRoot; divisor++) {
    if (number % divisor == 0) { // If true, number is not prime
      isPrime = false; // Set isPrime to false
      break; // Exit the for loop
    }
  }
...
```

Now we turn our attention to analyzing the complexity of this program. Since it takes $\sqrt{i}$ steps in the `for` loop (lines 21–27) to check whether number i is prime, the algorithm takes $\sqrt{2} + \sqrt{3} + \sqrt{4} + \cdots + \sqrt{n}$ steps to find all the prime numbers less than or equal to n. Observe that

$$\sqrt{2} + \sqrt{3} + \sqrt{4} + \cdots + \sqrt{n} \leq n\sqrt{n}$$

Therefore, the time complexity for this algorithm is $O\left(n\sqrt{n}\right)$.

To determine whether i is prime, the algorithm checks whether `2`, `3`, `4`, `5`, . . . , or $\sqrt{i}$ divides i. This algorithm can be further improved. In fact, you need to check only whether the prime numbers from 2 to $\sqrt{i}$ are possible divisors for i.

We can prove that if i is not prime, there must exist a prime number p such that $i = pq$ and $p \leq q$. Here is the proof. Assume i is not prime; let p be the smallest factor of i. p must be prime, otherwise, p has a factor k with $2 \leq k < p$. k is also a factor of i, which contradicts that p be the smallest factor of i. Therefore, if i is not prime, you can find a prime number from `2` to $\sqrt{i}$ that divides i. This leads to a more efficient algorithm for finding all prime numbers up to `n`, as given in Listing 22.6.

LISTING 22.6 `EfficientPrimeNumbers.java`

```
import java.util.Scanner;

public class EfficientPrimeNumbers {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Find all prime numbers <= n, enter n: ");
    int n = input.nextInt();

    // A list to hold prime numbers
    java.util.List<Integer> list =
      new java.util.ArrayList<>();

    final int NUMBER_PER_LINE = 10; // Display 10 per line
    int count = 0; // Count the number of prime numbers
    int number = 2; // A number to be tested for primeness
    int squareRoot = 1; // Check whether number <= squareRoot

    System.out.println("The prime numbers are \n");

    // Repeatedly find prime numbers
    while (number <= n) {
      // Assume the number is prime
      boolean isPrime = true; // Is the current number prime?

      if (squareRoot * squareRoot < number) squareRoot++;

      // Test whether number is prime
```

check prime

```
      for (int k = 0; k < list.size()
                        && list.get(k) <= squareRoot; k++) {
        if (number % list.get(k) == 0) {  // If true, not prime
          isPrime = false; // Set isPrime to false
          break; // Exit the for loop
        }
      }

      // Print the prime number and increase the count
      if (isPrime) {
```

increase count

```
        count++; // Increase the count
        list.add(number); // Add a new prime to the list
        if (count % NUMBER_PER_LINE == 0) {
          // Print the number and advance to the new line
          System.out.println(number);
        }
        else
          System.out.print(number + " ");
      }

      // Check whether the next number is prime
```

check next number

```
      number++;
    }

    System.out.println("\n" + count +
      " prime(s) less than or equal to " + n);
  }
}
```

```
Find all prime numbers <= n, enter n: 1000 ↵Enter
The prime numbers are:
      2      3      5      7     11     13     17     19     23     29
     31     37     41     43     47     53     59     61     67     71
...
...
168 prime(s) less than or equal to 1000
```

Let $\pi\,(i)$ denote the number of prime numbers less than or equal to i. The primes under `20` are `2`, `3`, `5`, `7`, `11`, `13`, `17`, and `19`. Therefore, $\pi\,(2)$ is `1`, $\pi\,(3)$ is `2`, $\pi\,(6)$ is `3`, and $\pi\,(20)$ is `8`. It has been proved that $\pi\,(i)$ is approximately $\frac{i}{\log i}$ (see primes.utm.edu/howmany.shtml).

For each number `i`, the algorithm checks whether a prime number less than or equal to $\sqrt{i}$ divides i. The number of the prime numbers less than or equal to $\sqrt{i}$ is

$$\frac{\sqrt{i}}{\log\sqrt{i}} = \frac{2\sqrt{i}}{\log i}$$

Thus, the complexity for finding all prime numbers up to `n` is

$$\frac{2\sqrt{2}}{\log 2} + \frac{2\sqrt{3}}{\log 3} + \frac{2\sqrt{4}}{\log 4} + \frac{2\sqrt{5}}{\log 5} + \frac{2\sqrt{6}}{\log 6} + \frac{2\sqrt{7}}{\log 7} + \frac{2\sqrt{8}}{\log 8} + \cdots + \frac{2\sqrt{n}}{\log n}$$

Since $\frac{\sqrt{i}}{\log i} < \frac{\sqrt{n}}{\log n}$ for $i < n$ and $n \geq 16$,

$$\frac{2\sqrt{2}}{\log 2} + \frac{2\sqrt{3}}{\log 3} + \frac{2\sqrt{4}}{\log 4} + \frac{2\sqrt{5}}{\log 5} + \frac{2\sqrt{6}}{\log 6} + \frac{2\sqrt{7}}{\log 7} + \frac{2\sqrt{8}}{\log 8} + \cdots + \frac{2\sqrt{n}}{\log n} < \frac{2n\sqrt{n}}{\log n}$$

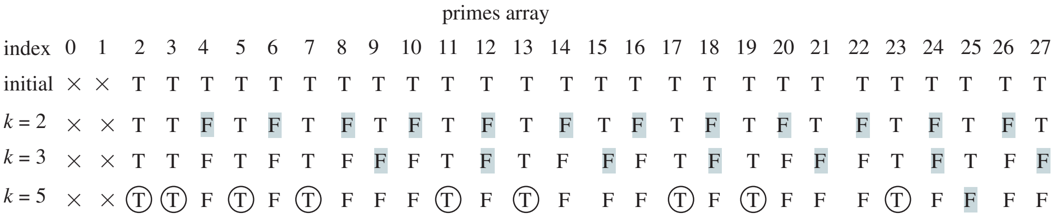
Therefore, the complexity of this algorithm is $O\left(\frac{n\sqrt{n}}{\log n}\right)$.

This algorithm is another example of dynamic programming. The algorithm stores the results of the subproblems in the array list and uses them later to check whether a new number is prime.

dynamic programming

Is there any algorithm better than $O\left(\frac{n\sqrt{n}}{\log n}\right)$? Let us examine the well-known Eratosthenes algorithm for finding prime numbers. Eratosthenes (276–194 B.C.) was a Greek mathematician who devised a clever algorithm, known as the *Sieve of Eratosthenes*, for finding all prime numbers $\leq n$. His algorithm is to use an array named `primes` of n Boolean values. Initially, all elements in `primes` are set `true`. Since the multiples of `2` are not prime, set `primes[2 * i]` to `false` for all $2 \leq i \leq n/2$, as shown in Figure 22.3. Since we don't care about `primes[0]` and `primes[1]`, these values are marked × in the figure.

Sieve of Eratosthenes



primes array

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
initial	×	×	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$k = 2$	×	×	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
$k = 3$	×	×	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F
$k = 5$	×	×	Ⓣ	Ⓣ	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	F	F	F

FIGURE 22.3 The values in `primes` are changed with each prime number `k`.

Since the multiples of `3` are not prime, set `primes[3 * i]` to `false` for all $3 \leq i \leq n/3$. Because the multiples of `5` are not prime, set `primes[5 * i]` to `false` for all $5 \leq i \leq n/5$. Note you don't need to consider the multiples of `4` because the multiples of `4` are also the multiples of `2`, which have already been considered. Similarly, multiples of `6`, `8`, and `9` need not be considered. You only need to consider the multiples of a prime number `k = 2`, `3`, `5`, `7`, `11`, . . . , and set the corresponding element in `primes` to `false`. Afterward, if `primes[i]` is still true, then `i` is a prime number. As shown in Figure 22.3, `2`, `3`, `5`, `7`, `11`, `13`, `17`, `19`, and `23` are prime numbers. Listing 22.7 gives the program for finding the prime numbers using the Sieve of Eratosthenes algorithm.

LISTING 22.7 SieveOfEratosthenes.java

```java
 1 import java.util.Scanner;
 2 
 3 public class SieveOfEratosthenes {
 4   public static void main(String[] args) {
 5     Scanner input = new Scanner(System.in);
 6     System.out.print("Find all prime numbers <= n, enter n: ");
 7     int n = input.nextInt();
 8 
 9     boolean[] primes = new boolean[n + 1]; // Prime number sieve
10 
11     // Initialize primes[i] to true
12     for (int i = 0; i < primes.length; i++) {
13       primes[i] = true;
14     }
15 
16     for (int k = 2; k <= n / k; k++) {
17       if (primes[k]) {
18         for (int i = k; i <= n / k; i++) {
19           primes[k * i] = false; // k * i is not prime
20         }
21       }
22     }
23 
24     final int NUMBER_PER_LINE = 10; // Display 10 per line
25     int count = 0; // Count the number of prime numbers found so far
26     // Print prime numbers
27     for (int i = 2; i < primes.length; i++) {
28       if (primes[i]) {
29         count++;
30         if (count % NUMBER_PER_LINE == 0)
31           System.out.printf("%7d\n", i);
32         else
33           System.out.printf("%7d", i);
34       }
35     }
36 
37     System.out.println("\n" + count +
38       " prime(s) less than or equal to " + n);
39   }
40 }
```

sieve

initialize sieve

nonprime

```
Find all prime numbers <= n, enter n: 1000 ↵Enter
The prime numbers are:
      2      3      5      7     11     13     17     19     23     29
     31     37     41     43     47     53     59     61     67     71
...
...
168 prime(s) less than or equal to 1000
```

Note `k <= n / k` (line 16). Otherwise, `k * i` would be greater than `n` (line 19). What is the time complexity of this algorithm?

For each prime number `k` (line 17), the algorithm sets `primes[k * i]` to `false` (line 19). This is performed `n / k - k + 1` times in the `for` loop (line 18). Thus, the complexity for finding all prime numbers up to `n` is

$$
\begin{aligned}
&\frac{n}{2} - 2 + 1 + \frac{n}{3} - 3 + 1 + \frac{n}{5} - 5 + 1 + \frac{n}{7} - 7 + 1 + \frac{n}{11} - 11 + 1 \ldots \\
&= O\left(\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} + \cdots\right) < O(n\pi(n)) \\
&= O\left(n\frac{\sqrt{n}}{\log n}\right)
\end{aligned}
$$

The number of items in the series is $\pi(n)$.

This upper bound $O\left(\frac{n\sqrt{n}}{\log n}\right)$ is very loose. The actual time complexity is much better than $O\left(\frac{n\sqrt{n}}{\log n}\right)$. The Sieve of Eratosthenes algorithm is good for a small `n` such that the array `primes` can fit in the memory.

Table 22.5 summarizes the complexity of these three algorithms for finding all prime numbers up to n.

TABLE 22.5 Comparisons of Prime-Number Algorithms

Algorithm	*Complexity*	*Description*
Listing 5.15	$O(n^2)$	Brute-force, checking all possible divisors
Listing 22.5	$O(n\sqrt{n})$	Checking divisors up to $\sqrt{n}$
Listing 22.6	$O\left(\frac{n\sqrt{n}}{\log n}\right)$	Checking prime divisors up to $\sqrt{n}$
Listing 22.7	$O\left(\frac{n\sqrt{n}}{\log n}\right)$	Sieve of Eratosthenes

Check Point

22.7.1 Prove that if n is not prime, there must exist a prime number p such that $p <= \sqrt{n}$ and p is a factor of n.

22.7.2 Describe how the sieve of Eratosthenes is used to find the prime numbers.

22.8 Finding the Closest Pair of Points Using Divide-and-Conquer

Key Point

This section presents efficient algorithms for finding the closest pair of points using divide-and-conquer.

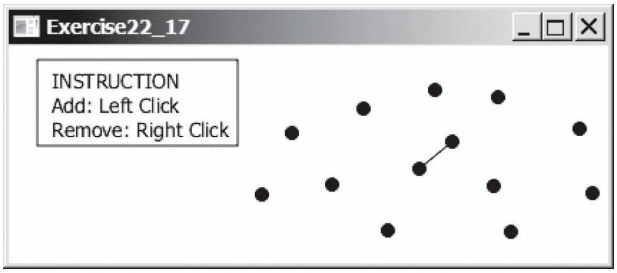
Pedagogical Note

Starting from this section, we will present interesting and challenging problems. It is time that you begin to study advanced algorithms to become a proficient programmer. We recommend that you study the algorithms and implement them in the exercises.

Given a set of points, the closest-pair problem is to find the two points that are nearest to each other. As shown in Figure 22.4, a line is drawn to connect the two nearest points in the closest-pair animation.

closest-pair animation on Companion Website

Section 8.6, Case Study: Finding the Closest Pair, presented a brute-force algorithm for finding the closest pair of points. The algorithm computes the distances between all pairs of



INSTRUCTION
Add: Left Click
Remove: Right Click

Figure 22.4 The closet-pair animation draws a line to connect the closest pair of points dynamically as points are added and removed interactively. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

points and finds the one with the minimum distance. Clearly, the algorithm takes $O(n^2)$ time. Can we design a more efficient algorithm? We can use an approach called *divide-and-conquer* to solve this problem efficiently in $O(n \log n)$ time.

divide-and-conquer

Algorithm Design Note

The *divide-and-conqure* approach divides the problem into subproblems, solves the subproblems, and then combines the solutions of the subproblems to obtain the solution for the entire problem. Unlike the dynamic programming approach, the subproblems in the divide-and-conquer approach don't overlap. A subproblem is like the original problem with a smaller size, so you can apply recursion to solve the problem. In fact, all the solutions for recursive problems follow the divide-and-conquer approach.

Listing 22.8 describes how to solve the closest pair problem using the divide-and-conquer approach.

Listing 22.8 Algorithm for Finding the Closest Pair

Step 1: Sort the points in increasing order of x-coordinates. For the points with the same x-coordinates, sort on y-coordinates. This results in a sorted list S of points.

Step 2: Divide S into two subsets, S_1 and S_2, of equal size using the midpoint in the sorted list. Let the midpoint be in S_1. Recursively find the closest pair in S_1 and S_2. Let d_1 and d_2 denote the distance of the closest pairs in the two subsets, respectively.

Step 3: Find the closest pair between a point in S_1 and a point in S_2 and denote their distance as d_3. The closest pair is the one with the distance $\min(d_1, d_2, d_3)$.

Selection sort takes $O(n^2)$ time. In Chapter 23, we will introduce merge sort and heap sort. These sorting algorithms take $O(n \log n)$ time. Step 1 can be done in $O(n \log n)$ time.

Step 3 can be done in $O(n)$ time. Let $d = \min(d_1, d_2)$. We already know that the closest-pair distance cannot be larger than d. For a point in S_1 and a point in S_2 to form the closest pair in S, the left point must be in `stripL` and the right point in `stripR`, as illustrated in Figure 22.5a.

For a point p in `stripL`, you need only consider a right point within the $d \times 2d$ rectangle, as shown in 22.5b. Any right point outside the rectangle cannot form the closest pair with p. Since the closest-pair distance in S_2 is greater than or equal to d, there can be at most six points in the rectangle. Thus, for each point in `stripL`, at most six points in `stripR` need to be considered.

For each point p in `stripL`, how do you locate the points in the corresponding $d \times 2d$ rectangle area in `stripR`? This can be done efficiently if the points in `stripL` and `stripR` are sorted in increasing order of their y-coordinates. Let `pointsOrderedOnY` be the list of the points sorted in increasing order of y-coordinates. `pointsOrderedOnY` can be obtained beforehand in the algorithm. `stripL` and `stripR` can be obtained from `pointsOrderedOnY` in Step 3 as given in Listing 22.9.

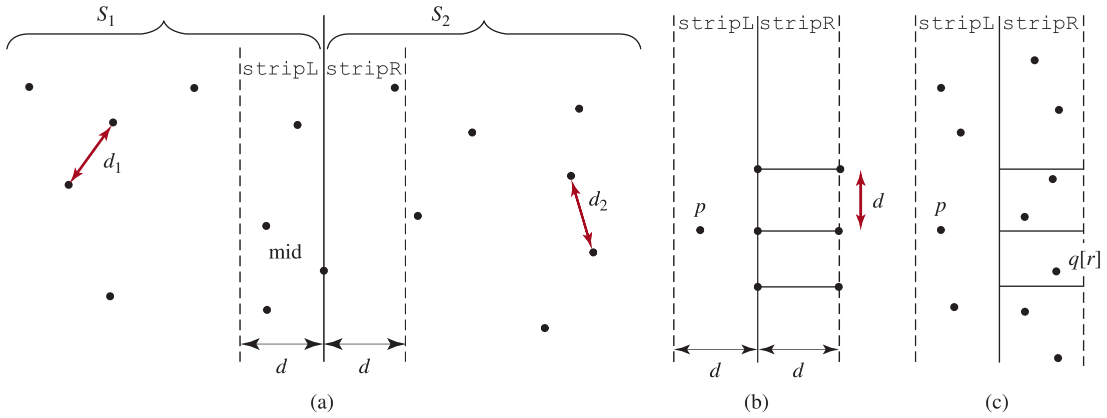


FIGURE 22.5 The midpoint divides the points into two sets of equal size.

LISTING 22.9 Algorithm for Obtaining stripL and stripR

```
for each point p in pointsOrderedOnY
  if (p is in S1 and mid.x – p.x <= d)
    append p to stripL;
  else if (p is in S2 and p.x – mid.x <= d)
    append p to stripR;
```

stripL

stripR

Let the points in **`stripL`** and **`stripR`** be $\{p_0, p_1, \ldots, p_k\}$ and $\{q_0, q_1, \ldots, q_t\}$, as shown in Figure 22.5c. The closest pair between a point in **`stripL`** and a point in **`stripR`** can be found using the algorithm described in Listing 22.10.

LISTING 22.10 Algorithm for Finding the Closest Pair in Step 3

```
 1  d = min(d1, d2);
 2  r = 0; // r is the index of a point in stripR
 3  for (each point p in stripL) {
 4    // Skip the points in stripR below p.y – d
 5    while (r < stripR.length && q[r].y <= p.y – d)
 6      r++;
 7
 8    let r1 = r;
 9    while (r1 < stripR.length && |q[r1].y – p.y| <= d) {
10      // Check if (p, q[r1]) is a possible closest pair
11      if (distance(p, q[r1]) < d) {
12        d = distance(p, q[r1]);
13        (p, q[r1]) is now the current closest pair;
14      }
15
16      r1 = r1 + 1;
17    }
18  }
```

update closest pair

The points in **`stripL`** are considered from $p_0, p_1, \ldots, p_k$ in this order. For a point **`p`** in **`stripL`**, skip the points in **`stripR`** that are below **`p.y – d`** (lines 5–6). Once a point is skipped, it will no longer be considered. The **`while`** loop (lines 9–17) checks whether **`(p, q[r1])`** is a possible closest pair. There are at most six such **`q[r1]`** pairs, because the distance between two points in **`stripR`** cannot be less than **`d`**. Thus, the complexity for finding the closest pair in Step 3 is $O(n)$.

Note Step 1 in Listing 22.8 is performed only once to presort the points. Assume all the points are presorted. Let $T(n)$ denote the time complexity for this algorithm. Thus,

$$T(n) = \overset{\text{Step 2}}{\overset{\downarrow}{2T(n/2)}} + \overset{\text{Step 3}}{\overset{\downarrow}{O(n)}} = O(n \log n)$$

Therefore, the closest pair of points can be found in $O(n \log n)$ time. The complete implementation of this algorithm is left as an exercise (see Programming Exercise 22.7).

Check Point

22.8.1 What is divide-and-conquer approach? Give an example.

22.8.2 What is the difference between divide-and-conquer and dynamic programming?

22.8.3 Can you design an algorithm for finding the minimum element in a list using divide-and-conquer? What is the complexity of this algorithm?

22.9 Solving the Eight Queens Problem Using Backtracking

Key Point

This section solves the Eight Queens problem using the backtracking approach.

backtracking

The Eight Queens problem is to find a solution to place a queen in each row on a chessboard such that no two queens can attack each other. The problem can be solved using recursion (see Programming Exercise 18.34). In this section, we will introduce a common algorithm design technique called *backtracking* for solving this problem.

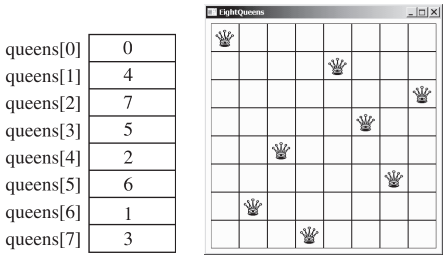
Algorithm Design Note

There are many possible candidates? How do you find a solution? The backtracking approach searches for a candidate solution incrementally, abandoning that option as soon as it determines that the candidate cannot possibly be a valid solution, and then looks for a new candidate.

You can use a two-dimensional array to represent a chessboard. However, since each row can have only one queen, it is sufficient to use a one-dimensional array to denote the position of the queen in the row. Thus, you can define the `queens` array as

```
int[] queens = new int[8];
```

Assign `j` to `queens[i]` to denote that a queen is placed in row `i` and column `j`. Figure 22.6 shows the contents of the `queens` array for the chessboard.



queens[0]	0
queens[1]	4
queens[2]	7
queens[3]	5
queens[4]	2
queens[5]	6
queens[6]	1
queens[7]	3

(a) (b)

FIGURE 22.6 `queens[i]` denotes the position of the queen in row `i`. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

search algorithm

The search starts from the first row with $k = 0$, where k is the index of the current row being considered. The algorithm checks whether a queen can be possibly placed in the jth column in the row for $j = 0, 1, \ldots, 7$, in this order. The search is implemented as follows:

- If successful, it continues to search for a placement for a queen in the next row. If the current row is the last row, a solution is found.
- If not successful, it backtracks to the previous row and continues to search for a new placement in the next column in the previous row.
- If the algorithm backtracks to the first row and cannot find a new placement for a queen in this row, no solution can be found.

Eight Queens animation on the Companion Website

To see how the algorithm works, go to http://liveexample.pearsoncmg.com/dsanimation/EightQueens.html.

Listing 22.11 gives the program that displays a solution for the Eight Queens problem.

Listing 22.11 EightQueens.java

```java
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.GridPane;

public class EightQueens extends Application {
  public static final int SIZE = 8; // The size of the chessboard
  // queens are placed at (i, queens[i])
  // -1 indicates that no queen is currently placed in the ith row
  // Initially, place a queen at (0, 0) in the 0th row
  private int[] queens = {-1, -1, -1, -1, -1, -1, -1, -1};

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    search(); // Search for a solution

    // Display chessboard
    GridPane chessBoard = new GridPane();
    chessBoard.setAlignment(Pos.CENTER);
    Label[][] labels = new Label[SIZE][SIZE];
    for (int i = 0; i < SIZE; i++)
      for (int j = 0; j < SIZE; j++) {
        chessBoard.add(labels[i][j] = new Label(), j, i);
        labels[i][j].setStyle("-fx-border-color: black");
        labels[i][j].setPrefSize(55, 55);
      }

    // Display queens
    Image image = new Image("image/queen.jpg");
    for (int i = 0; i < SIZE; i++)
      labels[i][queens[i]].setGraphic(new ImageView(image));

    // Create a scene and place it in the stage
    Scene scene = new Scene(chessBoard, 55 * SIZE, 55 * SIZE);
    primaryStage.setTitle("EightQueens"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /** Search for a solution */
  private boolean search() {
    // k - 1 indicates the number of queens placed so far
    // We are looking for a position in the kth row to place a queen
    int k = 0;
    while (k >= 0 && k < SIZE) {
      // Find a position to place a queen in the kth row
      int j = findPosition(k);
      if (j < 0) {
        queens[k] = -1;
        k--; // Backtrack to the previous row
      }  else {
```

queen positions

search for solution

create cells

set queen image

find a column

backtrack

place a queen
search the next row

```
56          queens[k] = j;
57          k++;
58        }
59      }
60
61      if (k == -1)
62        return false; // No solution
63      else
64        return true; // A solution is found
65    }
66
67    public int findPosition(int k) {
68      int start = queens[k] + 1; // Search for a new placement
69
70      for (int j = start; j < SIZE; j++) {
71        if (isValid(k, j))
72          return j; // (k, j) is the place to put the queen now
73      }
74
75      return -1;
76    }
77
78    /** Return true if a queen can be placed at (row, column) */
79    public boolean isValid(int row, int column) {
80      for (int i = 1; i <= row; i++)
81        if (queens[row - i] == column // Check column
82          || queens[row - i] == column - i // Check upleft diagonal
83          || queens[row - i] == column + i)  // Check upright diagonal
84          return false; // There is a conflict
85      return true; // No conflict
86    }
87  }
```

The program invokes **search()** (line 19) to search for a solution. Initially, no queens are placed in any rows (line 15). The search now starts from the first row with **k = 0** (line 48) and finds a place for the queen (line 51). If successful, place it in the row (line 56) and consider the next row (line 57). If not successful, backtrack to the previous row (lines 53–54).

The **findPosition(k)** method searches for a possible position to place a queen in row **k** starting from **queen[k] + 1** (line 68). It checks whether a queen can be placed at **start**, **start + 1**, . . . , and **7**, in this order (lines 70–73). If possible, return the column index (line 72); otherwise, return **-1** (line 75).

The **isValid(row, column)** method is called to check whether placing a queen at the specified position causes a conflict with the queens placed earlier (line 71). It ensures that no queen is placed in the same column (line 81), in the upper-left diagonal (line 82), or in the upper-right diagonal (line 83), as shown in Figure 22.7.

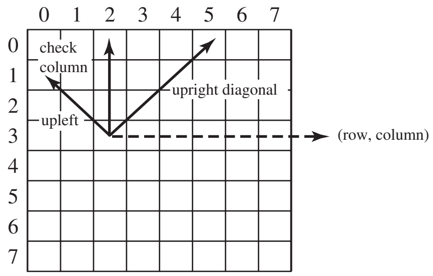


FIGURE 22.7 Invoking **isValid(row, column)** checks whether a queen can be placed at (row, column).

22.9.1 What is backtracking? Give an example.

22.9.2 If you generalize the Eight Queens problem to the *n*-Queens problem in an *n*-by-*n* chessboard, what will be the complexity of the algorithm?

Check Point

22.10 Computational Geometry: Finding a Convex Hull

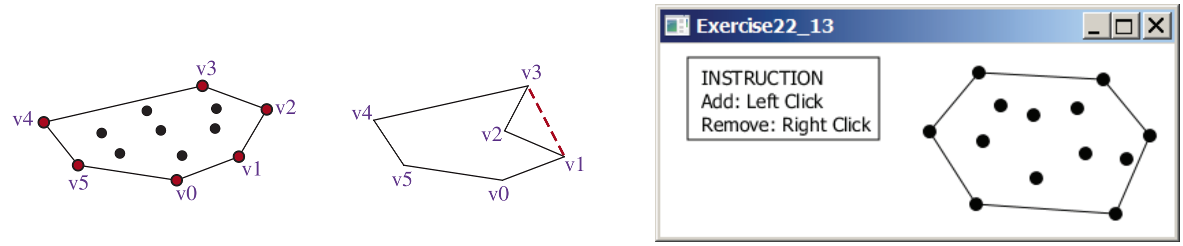
This section presents efficient geometric algorithms for finding a convex hull for a set of points.

Key Point

Computational geometry is to study the algorithms for geometrical problems. It has applications in computer graphics, games, pattern recognition, image processing, robotics, geographical information systems, and computer-aided design and manufacturing. Section 22.8 presented a geometrical algorithm for finding the closest pair of points. This section introduces geometrical algorithms for finding a convex hull.

convex hull

Given a set of points, a *convex hull* is the smallest convex polygon that encloses all these points, as shown in Figure 22.8a. A polygon is convex if every line connecting two vertices is inside the polygon. For example, the vertices v0, v1, v2, v3, v4, and v5 in Figure 22.8a form a convex polygon, but not in Figure 22.8b, because the line that connects v3 and v1 is not inside the polygon.



INSTRUCTION
Add: Left Click
Remove: Right Click

(a) A convex hull (b) A nonconvex polygon (c) Convex hull animation

Figure 22.8 A convex hull is the smallest convex polygon that contains a set of points. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

A convex hull has many applications in game programming, pattern recognition, and image processing. Before we introduce the algorithms, it is helpful to get acquainted with the concept using an interactive tool from Listing.pearsoncmg.com/dsanimation/ConvexHull.html, as shown in Figure 22.8c. This tool allows you to add and remove points and displays the convex hull dynamically.

convex hull animation on the Companion Website

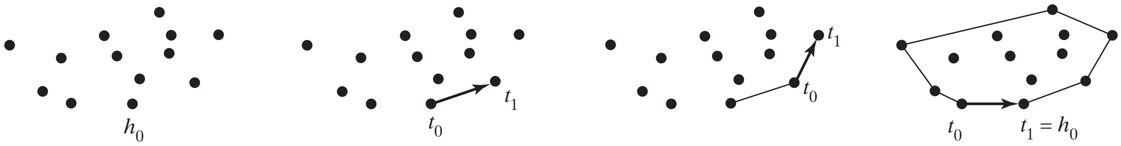
Many algorithms have been developed to find a convex hull. This section introduces two popular algorithms: the gift-wrapping algorithm and Graham's algorithm.

22.10.1 Gift-Wrapping Algorithm

An intuitive approach called the *gift-wrapping algorithm* works as described in Listing 22.12.

Listing 22.12 Finding a Convex Hull Using Gift-Wrapping Algorithm

```
Step 1: Given a list of points S, let the points in S be labeled
$s_0$, $s_1$, . . . , $s_k$. Select the rightmost lowest point S. As shown in
Figure 22.9a, $h_0$ is such a point. Add $h_0$ to list H. (H is initially emp-
ty. H will hold all points in the convex hull after the algorithm is
finished.) Let $t_0$ be $h_0$.
```



(a) Step 1 (b) Step 2 (c) Repeat Step 2 (d) *H* is found

FIGURE 22.9 (a) h_0 is the rightmost lowest point in S. (b) Step 2 finds point t_1. (c) A convex hull is expanded repeatedly. (d) A convex hull is found when t_1 becomes h_0.

```
Step 2: Let $t_1$ be $s_0$.
        For every point p in S,
          if p is on the right side of the direct line from $t_0$ to $t_1$, then
            let $t_1$ be p.
(After Step 2, no points lie on the right side of the direct line from $t_0$
to $t_1$, as shown in Figure 22.9b.)
Step 3: If $t_1$ is $h_0$ (see Figure 22.9d), the points in H form a convex
hull for S. Otherwise, add $t_1$ to H, let $t_0$ be $t_1$, and go back to Step 2
(see Figure 22.9c).
```

correctness of the algorithm

The convex hull is expanded incrementally. The correctness is supported by the fact that no points lie on the right side of the direct line from t_0 to t_1 after Step 2. This ensures that every line segment with two points in *S* falls inside the polygon.

time complexity of the algorithm

Finding the rightmost lowest point in Step 1 can be done in $O(n)$ time. Whether a point is on the left side of a line, right side, or on the line can be determined in $O(1)$ time (see Programming Exercise 3.32). Thus, it takes $O(n)$ time to find a new point t_1 in Step 2. Step 2 is repeated *h* times, where *h* is the size of the convex hull. Therefore, the algorithm takes $O(hn)$ time. In the worst-case, *h* is *n*.

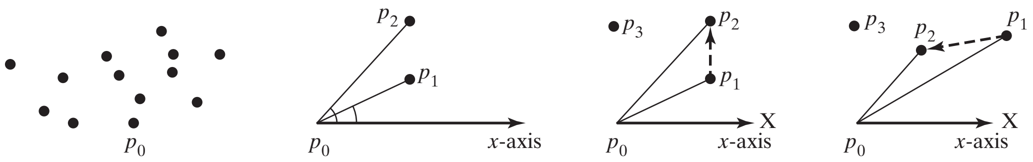
The implementation of this algorithm is left as an exercise (see Programming Exercise 22.9).

22.10.2 Graham's Algorithm

A more efficient algorithm was developed by Ronald Graham in 1972, as given in Listing 22.13.

LISTING 22.13 Finding a Convex Hull Using Graham's Algorithm

```
Step 1: Given a list of points S, select the rightmost lowest point and
name it $p_0$. As shown in Figure 22.10a, $p_0$ is such a point.
```



(a) Step 1 (b) Step 2 (c) p_3 into *H* (d) p_2 off *H*

FIGURE 22.10 (a) p_0 is the rightmost lowest point in S. (b) Points are sorted by their angles. (c–d) A convex hull is discovered incrementally.

```
Step 2: Sort the points in S angularly along the x-axis with $p_0$ as the
center, as shown in Figure 22.10b. If there is a tie and two points have
the same angle, discard the one that is closer to $p_0$. The points in S are
now sorted as $p_0$, $p_1$, $p_2$, . . . , $p_{n-1}$.
Step 3: Push $p_0$, $p_1$, and $p_2$ into stack H. (After the algorithm finishes,
H contains all the points in the convex hull.)
```

```
Step 4:
     i = 3;
     while (i < n) {
       Let $t_1$ and $t_2$ be the top first and second element in stack H;
       if ($p_i$ is on the left side of the direct line from $t_2$ to $t_1$) {
         Push $p_i$ to H;
         i++; // Consider the next point in S.
       }
       else
         Pop the top element off stack H.
     }
Step 5: The points in H form a convex hull.
```

The convex hull is discovered incrementally. Initially, p_0, p_1, and p_2 form a convex hull. Consider p_3. p_3 is outside of the current convex hull since points are sorted in increasing order of their angles. If p_3 is strictly on the left side of the line from p_1 to p_2 (see Figure 22.10c), push p_3 into H. Now p_0, p_1, p_2, and p_3 form a convex hull. If p_3 is on the right side of the line from p_1 to p_2 (see Figure 22.10d), pop p_2 out of H and push p_3 into H. Now p_0, p_1, and p_3 form a convex hull and p_2 is inside this convex hull. You can prove by induction that all the points in H in Step 5 form a convex hull for all the points in the input list S.

correctness of the algorithm

Finding the rightmost lowest point in Step 1 can be done in $O(n)$ time. The angles can be computed using trigonometry functions. However, you can sort the points without actually computing their angles. Observe p_2 would make a greater angle than p_1 if and only if p_2 lies on the left side of the line from p_0 to p_1. Whether a point is on the left side of a line can be determined in $O(1)$ time, as shown in Programming Exercise 3.32. Sorting in Step 2 can be done in $O(n \log n)$ time using the merge-sort or heap-sort algorithms that will be introduced in Chapter 23. Step 4 can be done in $O(n)$ time. Therefore, the algorithm takes $O(n \log n)$ time.

time complexity of the algorithm

The implementation of this algorithm is left as an exercise (see Programming Exercise 22.11).

Check Point

22.10.1 What is a convex hull?

22.10.2 Describe the gift-wrapping algorithm for finding a convex hull. Should list `H` be implemented using an `ArrayList` or a `LinkedList`?

22.10.3 Describe Graham's algorithm for finding a convex hull. Why does the algorithm use a stack to store the points in a convex hull?

22.11 String Matching

Key Point

This section presents the brute force, Boyer-Moore, and Knuth-Morris-Pratt algorithms for string matching.

String matching is to find a match for a substring in a string. The string is commonly known as the text and the substring is called a pattern. String matching is a common task in computer programming. The `String` class has the `text.contains(pattern)` method to test if a pattern is in a text and the `text.indexOf(pattern)` method to return the index of the first matching of the pattern in the text. A lot of research has been done to find efficient algorithms for string matching. This section presents three algorithms: the brute force algorithm, the Boyer-Moore algorithm, and the Knuth-Morris-Pratt algorithm.

The brute force algorithm is simply to compare the pattern with every possible substring in the text. Assume the length of text and pattern are `n` and `m`, respectively. The algorithm can be described as follows:

```
for i from 0 to n – m {
  test if pattern matches text[i .. i + m – 1]
}
```

Here text[`i ... j`] represents a substring in text from index `i` and index `j`. For an animation of this algorithm, see https://liveexample.pearsoncmg.com/dsanimation/StringMatch.html.

Listing 22.14 gives an implementation for the brute-force algorithm.

LISTING 22.14 StringMatch.java

```
 1  public class StringMatch {
 2    public static void main(String[] args) {
 3      java.util.Scanner input = new java.util.Scanner(System.in);
 4      System.out.print("Enter a string text: ");
 5      String text = input.nextLine();
 6      System.out.print("Enter a string pattern: ");
 7      String pattern = input.nextLine();
 8
 9      int index = match(text, pattern);
10      if (index >= 0)
11        System.out.println("matched at index "+ index);
12      else
13        System.out.println("unmatched");
14    }
15
16    // Return the index of the first match. -1 otherwise.
17    public static int match(String text, String pattern) {
18      for (int i = 0; i < text.length() - pattern.length() + 1; i++) {
19        if (isMatched(i, text, pattern))
20          return i;
21      }
22
23      return -1;
24    }
25
26    // Test if pattern matches text starting at index i
27    private static boolean isMatched(int i, String text,
28        String pattern) {
29      for (int k = 0; k < pattern.length(); k++) {
30        if (pattern.charAt(k) != text.charAt(i + k)) {
31          return false;
32        }
33      }
34
35      return true;
36    }
37  }
```

The `match(text, pattern)` method (lines 17–24) tests whether `pattern` matches a substring in `text`. The `isMatched(i, text, pattern)` method (lines 27–36) tests whether `pattern` matches `text[i ... i + m - 1]` starting at index `i`.

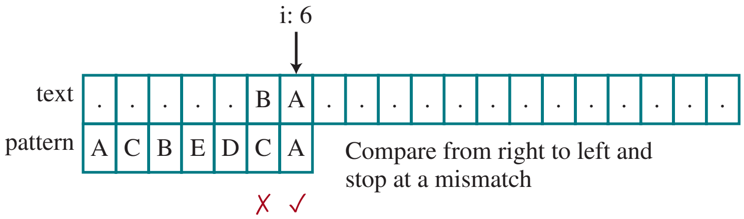
Clearly, the algorithm takes `O(nm)` time, since testing whether `pattern` matches `text[i ... i + m - 1]` takes `O(m)` time.

22.11.1 The Boyer-Moore Algorithm

The brute force algorithm searches for a match of the pattern in the text by examining all alignments. This is not necessary. The Boyer-Moore algorithm finds a matching by comparing the pattern with a substring in the text from right to left. If a character in the text does not match the one in the pattern and this character is not in the remaining part of the pattern, you can slide the pattern all the way passing this character. This can be best illustrated using the animation at https://liveexample.pearsoncmg.com/dsanimation/StringMatchBoyerMoore.html:

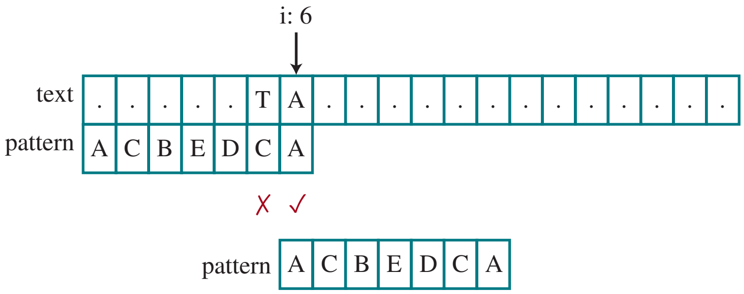
The Boyer-Moore algorithm can be described as follows:

```
i = m – 1;
while i <= n – 1
  Compare pattern with text[i – (m – 1) .. i]
  from right to left one by one, as shown in Figure 22.11.
  If they all match, done. Otherwise, let text[k] be the first one
  that does not match the corresponding character in pattern.
  Consider two cases:
  Case 1: If text[k] is not in the remaining pattern, slide the pat-
  tern passing text[k], as shown in Figure 22.12. Set i = k + m;
  Case 2: If text[k] is in pattern, find the last character, say
  pattern[j] in pattern that matches text[k] and slide the pattern
  right to align pattern[j] with text[k], as shown in Figure 22.13.
  Set i = k + m – j – 1.
```



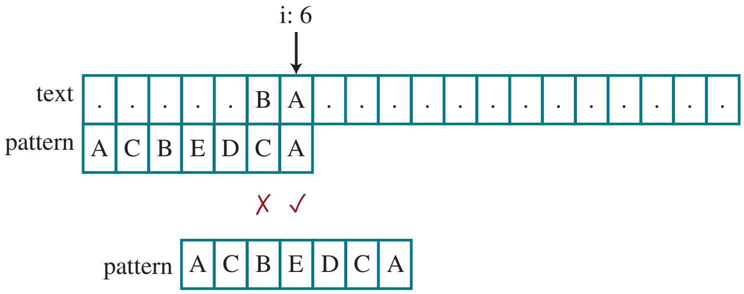
Compare from right to left and stop at a mismatch

FIGURE 22.11 Test if the pattern matches a substring by comparing the characters from right to left and stop at a mismatch.



Slide the pattern

FIGURE 22.12 Since T is not in the remaining of the pattern, slide the pattern passing T and start the next test.



Slide the pattern

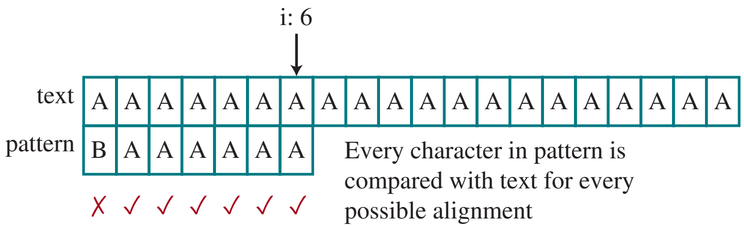
FIGURE 22.13 B does not match C. B first matches pattern[2] in the remaining part of the pattern from right to left, slide the pattern to align B in the text with pattern[2].

Listing 22.15 gives an implementation for the Boyer-Moore algorithm.

LISTING 22.15 StringMatchBoyerMoore.java

```java
public class StringMatchBoyerMoore {
  public static void main(String[] args) {
    java.util.Scanner input = new java.util.Scanner(System.in);
    System.out.print("Enter a string text: ");
    String text = input.nextLine();
    System.out.print("Enter a string pattern: ");
    String pattern = input.nextLine();

    int index = match(text, pattern);
    if (index >= 0)
      System.out.println("matched at index "+ index);
    else
      System.out.println("unmatched");
  }

  // Return the index of the first match. -1 otherwise.
  public static int match(String text, String pattern) {
    int i = pattern.length() - 1;
    while (i < text.length()) {
      int k = i;
      int j = pattern.length() - 1;
      while (j >= 0) {
        if (text.charAt(k) == pattern.charAt(j)) {
          k--; j--;
        }
        else {
          break;
        }
      }

      if (j < 0)
        return i = pattern.length() + 1; // A match found

      int u = findLastIndex(text.charAt(k), j - 1, pattern);
      if (u >= 0) { // text[k] is in the remaining part of the pattern
        i = k + pattern.length() - 1 - u;
      }
      else { // text[k] is in the remaining part of the pattern
        i = k + pattern.length();
      }
    }

    return -1;
  }

  // Return the index of the last element in pattern[0 .. j]
  // that matches ch. -1 otherwise.
  private static int findLastIndex(char ch, int j, String pattern) {
    for (int  k = j; k >= 0; k--) {
      if (ch == pattern.charAt(k)) {
        return k;
      }
    }

    return - 1;
  }
}
```

The `match(text, pattern)` method (lines 17–44) tests whether `pattern` matches a substring in `text. i` indicates the last index of a substring. It starts with `i = pattern.length() – 1` (line 21) and compares `text[i]` with `pattern[j]`, `text[i–1]` with `pattern[j–1]`, and so on backward (lines 22–29). If `j < 0`, a match is found (lines 31–32). Otherwise, find the index of the last matching element for `text[k]` in `pattern[0 .. j-1]` using the `findLastIndex` method (line 34). If the `index` is >= 0, set `k + m – 1 – index` (line 36), where m is `pattern.length()`. Otherwise, set `k + m` to `i` (line 39).
In the worst case, the Boyer-Moore algorithm takes `O(nm)` time. An example that achieves the worst case is shown in Figure 22.14.



Every character in pattern is compared with text for every possible alignment

FIGURE 22.14 The text is all As and the pattern is BAAAAAA. Every character in the pattern is compared with a character in the text for each possible alignment.

However, on the average time, the Boyer-Moore algorithm is efficient because the algorithm is often able to skip a large portion of text. There are several variations of the Boyer-Moore algorithm. We presented a simplified version in this section.

22.11.2 The Knuth-Morris-Pratt Algorithm

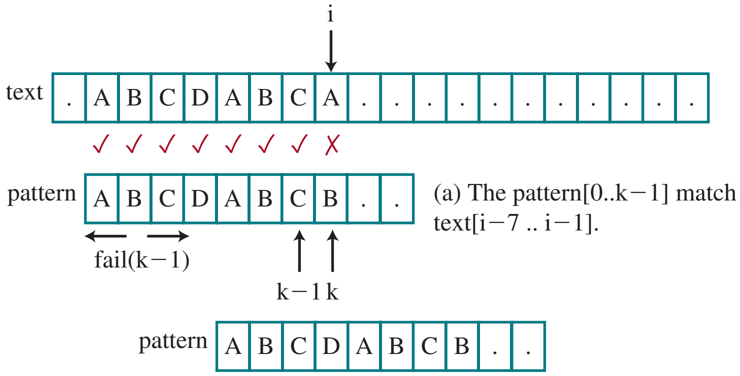
The Knuth-Morris-Pratt (KMP) algorithm is efficient. It achieves `O(m + n)` in the worst case. It is optimal because every character in the text and in the pattern must be checked at least once in the worst case. In the brute force or the Boyer-Moore algorithm, once a mismatch is found, the algorithm restarts to search for the next possible match by shifting the pattern one position to the right for the brute force algorithm and possibly multiple positions to the right for the Boyer-Moore algorithm. In doing so, the successful match of characters prior to the mismatch is ignored. The KMP algorithm takes consideration of the successful matches to find the maximum number of positions to shift in the pattern before continuing next search.

To find the maximum number of positions to shift in the pattern, we first define a failure function `fail(k)` as the length of the longest prefix of pattern that is a suffix of `pattern[0 .. k]`. The failure function can be precomputed for a given pattern. The failure function is actually an array with m elements. Suppose the pattern is `ABCABCDABC`. The failure functions for this pattern are shown in Figure 22.15:

k	0	1	2	3	4	5	6	7	8	9
pattern	A	B	C	A	B	C	D	A	B	C
fail	0	0	0	1	2	3	0	1	2	3

FIGURE 22.15 Failure function `fail[k]` is the length of the longest prefix that matches the suffix in `pattern[0..k]`.

For example, `fail[5]` is `3`, because `ABC` is the longest prefix that is suffix for `ABCABC`. `fail(7)` is 1, because `A` is the longest prefix that is suffix for `ABCABCDA`. When comparing the text and the pattern, once a mismatch is found at index `k` in the pattern, you can shift the pattern to align the pattern at index `fail[k-1]` with the text, as shown in Figure 22.16.



(a) The pattern[0..k−1] match text[i−7 .. i−1].

(b) Slide the pattern to align pattern[fail[k−1]] with text[i].

FIGURE 22.16 Upon a mismatch at `text[i]`, the pattern is shifted right to align the first `fail[k–1]` elements in the prefix with `text[i–1]`.

The KMP algorithm can be described as follows:

```
Step 1: First we precompute the failure functions. Now start with i = 0
and k = 0.
Step 2: Compare text[i] with pattern[k]. Consider two cases:
          Case 1 text[i] is equal to pattern[k]: if k is m-1, a matching
          is found and return i – m + 1. Otherwise, increase i and k by
          1.
          Case 2 text[i] is not equal to pattern[k]: if k > 0, shift the
          longest prefix that matches the suffix in pattern[k – 1] so
          that the last character in the prefix is aligned with text[i –
          1] by setting k = fail[k – 1], else increase i by 1.
Step 3: If i < n, repeat Step 2.
```

Now let us turn the attention to computing the failure functions. This can be done by comparing pattern with itself as follows:

```
Step 1: The failure function is an array with m elements. Initially, set
all the elements to 0. We start with i = 1 and k = 0.
Step 2: Compare pattern[i] with pattern[k]. Consider two cases:
          Case 1 pattern[i] == pattern[k]: fail[i] = k + 1. Increase i
          and k by 1.
          Case 2 pattern[i] != pattern[k]: if k > 0, set k = fail[k – 1],
          else increase i by 1.
Step 3: If i < m, repeat Step 2. Note that k indicates the length of the
longest prefix in pattern[0..i–1] if pattern[i] == pattern[k].
```

This is a sophisticated algorithm. The following animations are effective to help you understand the algorithms: https://liveexample.pearsoncmg.com/dsanimation/StringMatchKMP.html and https://liveexample.pearsoncmg.com/dsanimation/StringMatchKMPFail.html

The following animation shows how to obtain the failure functions.

Listing 22.16 gives an implementation of the KMP algorithm.

LISTING 12.16 StringMatchKMP.java

```java
public class StringMatchKMP {
  public static void main(String[] args) {
    java.util.Scanner input = new java.util.Scanner(System.in);
    System.out.print("Enter a string text: ");
    String text = input.nextLine();
    System.out.print("Enter a string pattern: ");
    String pattern = input.nextLine();

    int index = match(text, pattern);
    if (index >= 0)
      System.out.println("matched at index "+ index);
    else
      System.out.println("unmatched");
  }

  // Return the index of the first match. -1 otherwise.
  public static int match(String text, String pattern) {
    int[] fail = getFailure(pattern);
    int i = 0; // Index on text
    int k = 0; // Index on pattern
    while (i < text.length()) {
      if (text.charAt(i) == pattern.charAt(k)) {
        if (k == pattern.length() - 1) {
          return i- pattern.length() + 1; // pattern matched
        }
        i++; // Compare the next pair of characters
        k++;
      }
      else {
        if (k > 0) {
          k = fail[k -1]; // Matching prefix position
        }
        else {
          i++; // No prefix
        }
      }
    }

    return -1;
  }

  // Compute failure function
  private static int[] getFailure(String pattern) {
    int[] fail = new int [pattern.length()];
    int i = 1;
    int k = 0;
    while (i < pattern.length()) {
      if (pattern.charAt(i) == pattern.charAt(k)) {
        fail[i] = k + 1;
        i++;
        k++;
      }
      else if (k > 0) {
        k = fail[k -1];
      }
56        else {
57          i++;
58        }
59      }
60
61      return fail;
62    }
63  }
```

The `match(text, pattern)` method (lines 17–40) tests whether `pattern` matches a substring in `text. i` indicates the current position in the text, which always moves forward. `k` indicates the current position in the pattern. If `text[i] == pattern[k]` (line 22), both `i` and `k` is incremented by `1` (lines 26–27). Otherwise, if `k > 0`, set `fail(k – 1)` to `k` so to slide the pattern to align `pattern[k]` with `text[i]` (line 31), else increase `i` by `1` (line 34).

The `getFailure(pattern)` method (lines 43–62) compares `pattern` with `pattern` to obtain, `fail[k]`, the length of the maximum prefix that is the suffix in `pattern[0..k]`. It initializes the array `fail` to zeros (line 44) and set `i` and `k` to `1` and `0`, respectively (lines 45–46). `i` indicates the current position in the first pattern, which always moves forward. `k` indicates the current maximum length of a possible prefix that is also a suffix in `pattern[0..i]`. If `pattern[i] == pattern[k]`, set `fail[i]` to `k + 1` (line 49) and increase both `i` and `k` by `1` (lines 50–51). Otherwise, if `k > 0`, set `k` to `fail[k – 1]` to slide the second pattern to align `pattern[i]` in the first pattern with `pattern[k]` in the second pattern (line 54), else increase `i` by `1` (line 57).

To analyze the running time, consider three cases:

Case 1: `text[i]` is equal to `pattern[k]`. `i` is moved forward one position.
Case 2: `text[i]` is not equal to `pattern[k]` and `k` is `0`. `i` is moved forward one position.
Case 3: `text[i]` is not equal to `pattern[k]` and `k > 0`. The pattern is moved at least one position forward.

In any case, either `i` is moved forward one position on the text or the pattern is shifted at least one position to the right. Therefore, the number of iterations in the while loop for the match method is at most `2n`. Similarly, the number of the iterations in the `getFailure` method is at most `2m`. Therefore the running time of the KMP algorithm is `O(n + m)`.

KEY TERMS

average-case analysis	840	dynamic programming approach	850
backtracking approach	864	exponential time	847
best-case input	840	growth rate	840
Big *O* notation	840	logarithmic time	846
brute force	851	quadratic time	843
constant time	841	space complexity	841
convex hull	867	time complexity	841
divide-and-conquer approach	862	worst-case input	840

CHAPTER SUMMARY

1. The *Big O notation* is a theoretical approach for analyzing the performance of an algorithm. It estimates how fast an algorithm's execution time increases as the input size increases, which enables you to compare two algorithms by examining their *growth rates*.

2. An input that results in the shortest execution time is called the *best-case* input, and one that results in the longest execution time is called the *worst-case* input. Best- and worst-case analyses are not representative, but worst-case analysis is very useful. You can be assured that the algorithm will never be slower than the worst case.

3. An *average-case analysis* attempts to determine the average amount of time among all possible input of the same size. Average-case analysis is ideal, but difficult to perform because for many problems, it is hard to determine the relative probabilities and distributions of various input instances.

4. If the time is not related to the input size, the algorithm is said to take *constant time* with the notation $O(1)$.

5. Linear search takes $O(n)$ time. An algorithm with the $O(n)$ time complexity is called a *linear algorithm* and it exhibits a linear growth rate. Binary search takes $O(\log n)$ time. An algorithm with the $O(\log n)$ time complexity is called a *logarithmic algorithm* and it exhibits a logarithmic growth rate.

6. The worst-time complexity for selection sort is $O(n^2)$. An algorithm with the $O(n^2)$ time complexity is called a *quadratic algorithm* and it exhibits a quadratic growth rate.

7. The time complexity for the Tower of Hanoi problem is $O(2^n)$. An algorithm with the $O(2^n)$ time complexity is called an *exponential algorithm*, and it exhibits an exponential growth rate.

8. A Fibonacci number at a given index can be found in $O(n)$ time using dynamic programming approach.

9. Dynamic programming is the process of solving subproblems, then combining the solutions of the subproblems to obtain an overall solution. The key idea behind dynamic programming is to solve each subproblem only once and store the results for subproblems for later use to avoid redundant computing of the subproblems.

10. Euclid's GCD algorithm takes $O(\log n)$ time.

11. All prime numbers less than or equal to n can be found in $O\left(\frac{n\sqrt{n}}{\log n}\right)$ time.

12. The closest pair can be found in $O(n \log n)$ time using the *divide-and-conquer approach*.

13. The divide-and-conquer approach divides the problem into subproblems, solves the subproblems, and then combines the solutions of the subproblems to obtain the solution for the entire problem. Unlike the dynamic programming approach, the subproblems in the divide-and-conquer approach don't overlap. A subproblem is like the original problem with a smaller size, so you can apply recursion to solve the problem.

14. The Eight Queens problem can be solved using backtracking.

15. The backtracking approach searches for a candidate solution incrementally, abandoning that option as soon as it determines the candidate cannot possibly be a valid solution, then looks for a new candidate.

16. A *convex hull* for a set of points can be found in $O(n^2)$ time using the gift-wrapping algorithm, and in $O(n \log n)$ time using the Graham's algorithm.

17. The brute force and Boyer-Moore string matching algorithms take O(nm) time and the KMP string matching algorithm takes O(n + m) time.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

***22.1** (*Maximum consecutive increasingly ordered substring*) Write a program that prompts the user to enter a string and displays the maximum consecutive increasingly ordered substring. Analyze the time complexity of your program. Here is a sample run:

```
Enter a string: abcabcdgabxy ↵Enter
Maximum consecutive substring is abcdg
```

```
Enter a string: abcabcdgabmnsxy ↵Enter
Maximum consecutive substring is abmnsxy
```

****22.2** (*Maximum increasingly ordered subsequence*) Write a program that prompts the user to enter a string and displays the maximum increasingly ordered subsequence of characters. Analyze the time complexity of your program. Here is a sample run:

```
Enter a string: Welcome ↵Enter
Maximum increasingly ordered subsequence is Welo
```

***22.3** (*Pattern matching*) Write an `O(n)` time program that prompts the user to enter two strings and tests whether the second string is a substring of the first string. *Suppose the neighboring characters in the string are distinct.* (Don't use the `indexOf` method in the `String` class.) Here is a sample run of the program:

```
Enter a string s1: Welcome to Java ↵Enter
Enter a string s2: come ↵Enter
matched at index 3
```

***22.4** (*Revise Boyer-Moore algorithm*) Revise the implementation for the Boyer-Moore algorithm in Listing 22.15 `StringMatchBoyerMoore.java` to test where a mismatch character is in the pattern in O(1) time using a set that consists of all the characters in the pattern. If the test is false, the algorithm can shift the pattern past the mismatched character.

```
Enter a string s1: Mississippi ↵Enter
Enter a string s2: sip ↵Enter
matched at index 6
```

***22.5** (*Same-number subsequence*) Write an $O(n)$ time program that prompts the user to enter a sequence of integers ending with `0` and finds the longest subsequence with the same number. Here is a sample run of the program:

```
Enter a series of numbers ending with 0:
2 4 4 8 8 8 8 2 4 4 0 ↵Enter
The longest same number sequence starts at index 3 with 4 values of 8
```

***22.6** (*Execution time for GCD*) Write a program that obtains the execution time for finding the GCD of every two consecutive Fibonacci numbers from the index 40 to index 45 using the algorithms in Listing 22.3 and Listing 22.4. Your program should print a table like this:

	40	41	42	43	44	45
Listing 22.3 GCD						
Listing 22.4 GCDEuclid						

(*Hint*: You can use the following code template to obtain the execution time.)

```
long startTime = System.currentTimeMillis();
perform the task;
long endTime = System.currentTimeMillis();
long executionTime = endTime - startTime;
```

****22.7** (*Closest pair of points*) Section 22.8 introduced an algorithm for finding the closest pair of points using a divide-and-conquer approach. Implement the algorithm to meet the following requirements:

- Define a class named `Pair` with the data fields `p1` and `p2` to represent two points and a method named `getDistance()` that returns the distance between the two points.
- Implement the following methods:

```
/** Return the distance of the closest pair of points */
public static Pair getClosestPair(double[][] points)

/** Return the distance of the closest pair of points */
public static Pair getClosestPair(Point2D[] points)

/** Return the distance of the closest pair of points
  * in pointsOrderedOnX[low..high]. This is a recursive
  * method. pointsOrderedOnX and pointsOrderedOnY are
  * not changed in the subsequent recursive calls.
  */
public static Pair distance(Point2D[] pointsOrderedOnX,
    int low, int high, Point2D[] pointsOrderedOnY)

/** Compute the distance between two points p1 and p2 */
public static double distance(Point2D p1, Point2D p2)

/** Compute the distance between points (x1, y1) and (x2, y2) */
public static double distance(double x1, double y1,
    double x2, double y2)
```

****22.8** (*All prime numbers up to* `10,000,000,000`) Write a program that finds all prime numbers up to `10,000,000,000`. There are approximately `455,052,511` such prime numbers. Your program should meet the following requirements:

- Your program should store the prime numbers in a binary data file, named **PrimeNumbers.dat**. When a new prime number is found, the number is appended to the file.
- To find whether a new number is prime, your program should load the prime numbers from the file to an array of the `long` type of size `10000`. If no number in the array is a divisor for the new number, continue to read the next `10000` prime numbers from the data file, until a divisor is found or all numbers in the file are read. If no divisor is found, the new number is prime.
- Since this program takes a long time to finish, you should run it as a batch job from a UNIX machine. If the machine is shut down and rebooted, your program should resume by using the prime numbers stored in the binary data file rather than start over from scratch.

****22.9** (*Geometry: gift-wrapping algorithm for finding a convex hull*) Section 22.10.1 introduced the gift-wrapping algorithm for finding a convex hull for a set of points. Implement the algorithm using the following method:

```
/** Return the points that form a convex hull */
public static ArrayList<Point2D> getConvexHull(double[][] s)
```

Point2D was introduced in Section 9.6.3.

Write a test program that prompts the user to enter the set size and the points, and displays the points that form a convex hull. Note that when you debug the code, you will discover that the algorithm overlooked two cases (1) when `t1 = t0` and (2) when there is a point that is on the same line from `t0` to `t1`. When either case happens, replace `t1` by point `p` if the distance from `t0` to `p` is greater than the distance from `t0` to `t1`. Here is a sample run:

```
How many points are in the set?  6 ↵Enter
Enter 6 points:  1 2.4 2.5 2 1.5 34.5 5.5 6 6 2.4 5.5 9 ↵Enter
The convex hull is
     (2.5, 2.0) (6.0, 2.4) (5.5, 9.0) (1.5, 34.5) (1.0, 2.4)
```

22.10 (*Number of prime numbers*) Programming Exercise 22.8 stores the prime numbers in a file named **PrimeNumbers.dat**. Write a program that finds the number of prime numbers that are less than or equal to `10`, `100`, `1,000`, `10,000`, `100,000`, `1,000,000`, `10,000,000`, `100,000,000`, `1,000,000,000`, and `10,000,000,000`. Your program should read the data from **PrimeNumbers.dat**.

****22.11** (*Geometry: Graham's algorithm for finding a convex hull*) Section 22.10.2 introduced Graham's algorithm for finding a convex hull for a set of points. Assume Java's coordinate system is used for the points. Implement the algorithm using the following method:

```
/** Return the points that form a convex hull */
public static ArrayList<MyPoint> getConvexHull(double[][] s)
```

`MyPoint` is a static inner class defined as follows:

```
private static class MyPoint implements Comparable<MyPoint> {
  double x, y;

  MyPoint rightMostLowestPoint;

  MyPoint(double x, double y) {
    this.x = x; this.y = y;
  }

  public void setRightMostLowestPoint(MyPoint p) {
    rightMostLowestPoint = p;
  }

  @Override
  public int compareTo(MyPoint o) {
    // Implement it to compare this point with point o
    // angularly along the x-axis with rightMostLowestPoint
    // as the center, as shown in Figure 22.10b. By implementing
    // the Comparable interface, you can use the Array.sort
    // method to sort the points to simplify coding.
  }
}
```

Write a test program that prompts the user to enter the set size and the points, and displays the points that form a convex hull. Here is a sample run:

```
How many points are in the set? 6 ↵Enter
Enter six points: 1 2.4 2.5 2 1.5 34.5 5.5 6 6 2.4 5.5 9 ↵Enter
The convex hull is
    (2.5, 2.0) (6.0, 2.4) (5.5, 9.0) (1.5, 34.5) (1.0, 2.4)
```

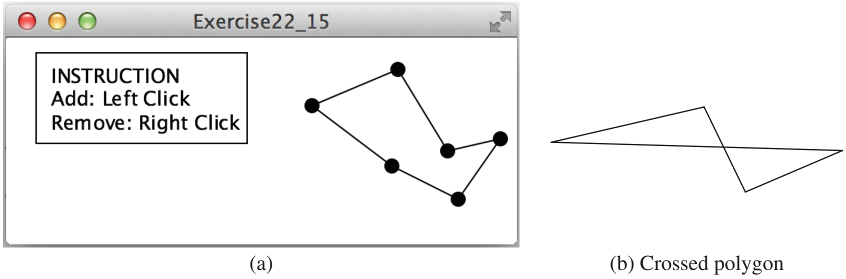
***22.12** (*Last 100 prime numbers*) Programming Exercise 22.8 stores the prime numbers in a file named **PrimeNumbers.dat.** Write an efficient program that reads the last `100` numbers in the file. (*Hint*: Don't read all numbers from the file. Skip all numbers before the last 100 numbers in the file.)

***22.13** (*Geometry: convex hull animation*) Programming Exercise 22.11 finds a convex hull for a set of points entered from the console. Write a program that enables the user to add or remove points by clicking the left or right mouse button and displays a convex hull, as shown in Figure 22.8c.

***22.14** (*Execution time for prime numbers*) Write a program that obtains the execution time for finding all the prime numbers less than 8,000,000, 10,000,000, 12,000,000, 14,000,000, 16,000,000, and 18,000,000 using the algorithms in Listings 22.5–22.7. Your program should print a table like this:

	8,000,000	10,000,000	12,000,000	14,000,000	16,000,000	18,000,000
Listing 22.5						
Listing 22.6						
Listing 22.7						

****22.15** (*Geometry: noncrossed polygon*) Write a program that enables the user to add or remove points by clicking the left or right mouse button and displays a noncrossed polygon that links all the points, as shown in Figure 22.17a. A polygon is crossed if two or more sides intersect, as shown in Figure 22.17b. Use the following algorithm to construct a polygon from a set of points:



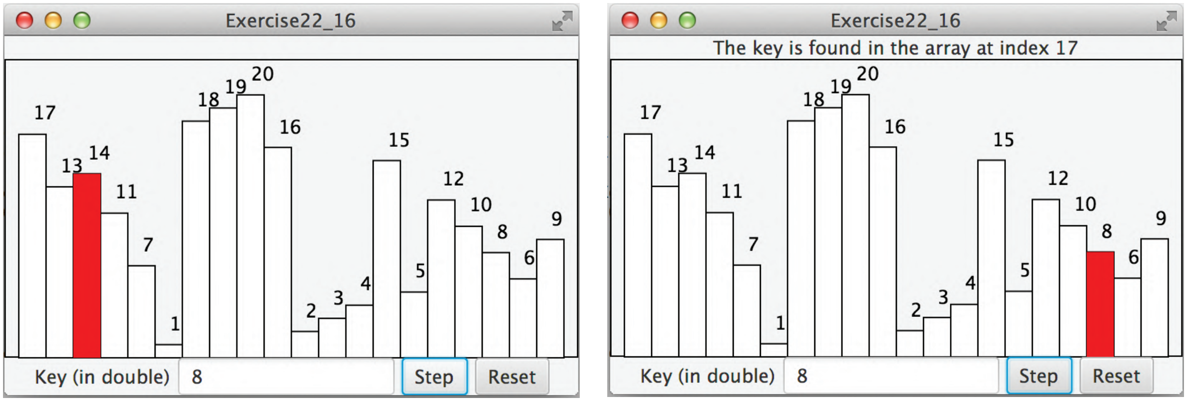
(a)

(b) Crossed polygon

FIGURE 22.17 (a) Programming Exercise 22.15 displays a noncrossed polygon for a set of points. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) Two or more sides intersect in a crossed polygon.

```
Step 1: Given a set of points S, select the rightmost lowest
point $p_0$ in the set S.
Step 2: Sort the points in S angularly along the x-axis with $p_0$
as the center. If there is a tie and two points have the same
angle, the one that is closer to $p_0$ is considered greater. The
points in S are now sorted as $p_0$, $p_1$, $p_2$, ..., $p_{n-1}$.
Step 3: The sorted points form a noncrossed polygon.
```

****22.16** (*Linear search animation*) Write a program that animates the linear search algorithm. Create an array that consists of 20 distinct numbers from 1 to 20 in a random order. The array elements are displayed in a histogram, as shown in Figure 22.18. You need to enter a search key in the text field. Clicking the *Step* button causes the program to perform one comparison in the algorithm and repaints the histogram with a bar indicating the search position. This button



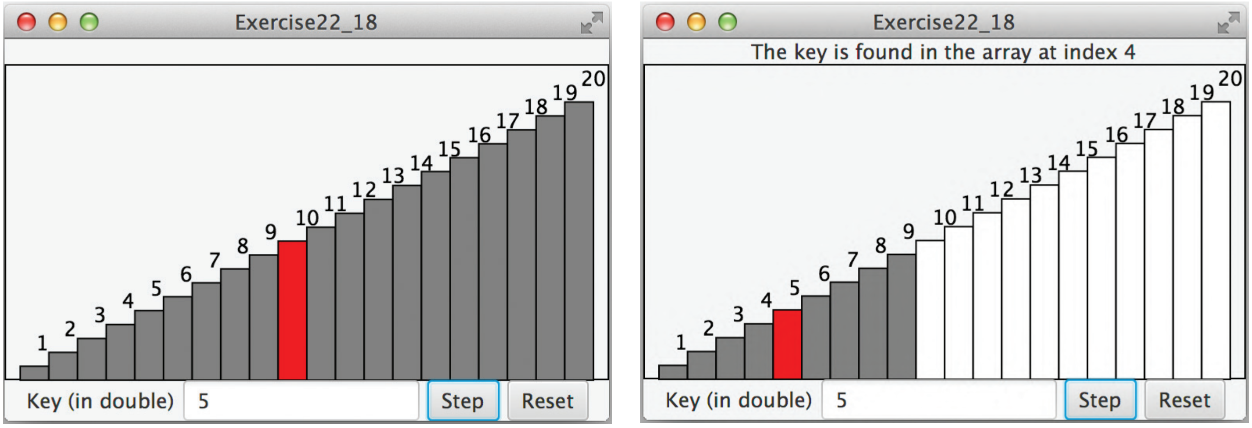
The key is found in the array at index 17

FIGURE 22.18 The program animates a linear search. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

also freezes the text field to prevent its value from being changed. When the algorithm is finished, display the status in the label at the top of the border pane to inform the user. Clicking the *Reset* button creates a new random array for a new start. This button also makes the text field editable.

****22.17** (*Closest-pair animation*) Write a program that enables the user to add/remove points by clicking the left/right mouse button and displays a line that connects the pair of nearest points, as shown in Figure 22.4.

****22.18** (*Binary search animation*) Write a program that animates the binary search algorithm. Create an array with numbers from 1 to 20 in this order. The array elements are displayed in a histogram, as shown in Figure 22.19. You need to enter a search key in the text field. Clicking the *Step* button causes the program to perform one comparison in the algorithm. Use a light-gray color to paint the bars for the numbers in the current search range, and use a black color to paint the bar indicating the middle number in the search range. The *Step* button also freezes the text field to prevent its value from being changed. When the algorithm is finished, display the status in a label at the top of a border pane. Clicking the *Reset* button enables a new search to start. This button also makes the text field editable.

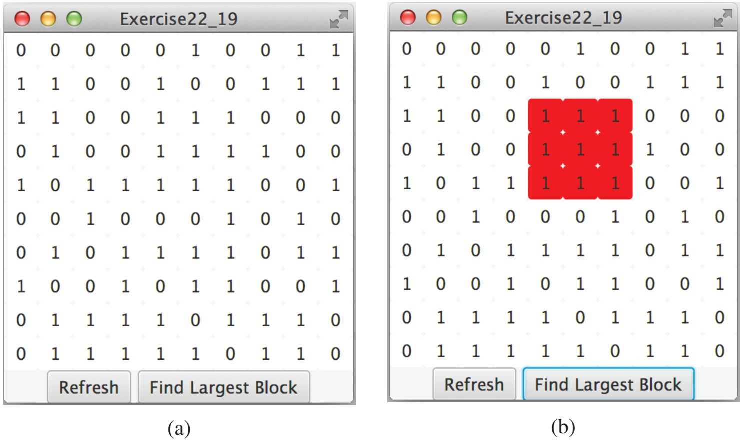


The key is found in the array at index 4

FIGURE 22.19 The program animates a binary search. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***22.19** (*Largest block*) The problem for finding a largest block is described in Programming Exercise 8.35. Design a dynamic programming algorithm for solving this problem in $O(n^2)$ time. Write a test program that displays a 10-by-10 square matrix, as shown in Figure 22.20a. Each element in the matrix is 0 or 1, randomly generated with a click of the *Refresh* button. Display each number centered in a text field. *Use a text field for each entry. Allow the user to change the entry value.* Click the *Find Largest Block* button to find a largest square submatrix that consists of 1s. Highlight the numbers in the block, as shown in Figure 22.20b.

*****22.20** (*Game: multiple Sudoku solutions*) The complete solution for the Sudoku problem is given in Supplement VI.A. A Sudoku problem may have multiple solutions. Modify Sudoku.java in Supplement VI.A to display the total number of solutions. Display two solutions if multiple solutions exist.



(a)

0	0	0	0	0	1	0	0	1	1
1	1	0	0	1	0	0	1	1	1
1	1	0	0	1	1	1	0	0	0
0	1	0	0	1	1	1	1	0	0
1	0	1	1	1	1	1	0	0	1
0	0	1	0	0	0	1	0	1	0
0	1	0	1	1	1	1	0	1	1
1	0	0	1	0	1	1	0	0	1
0	1	1	1	1	0	1	1	1	0
0	1	1	1	1	1	0	1	1	0

(b)

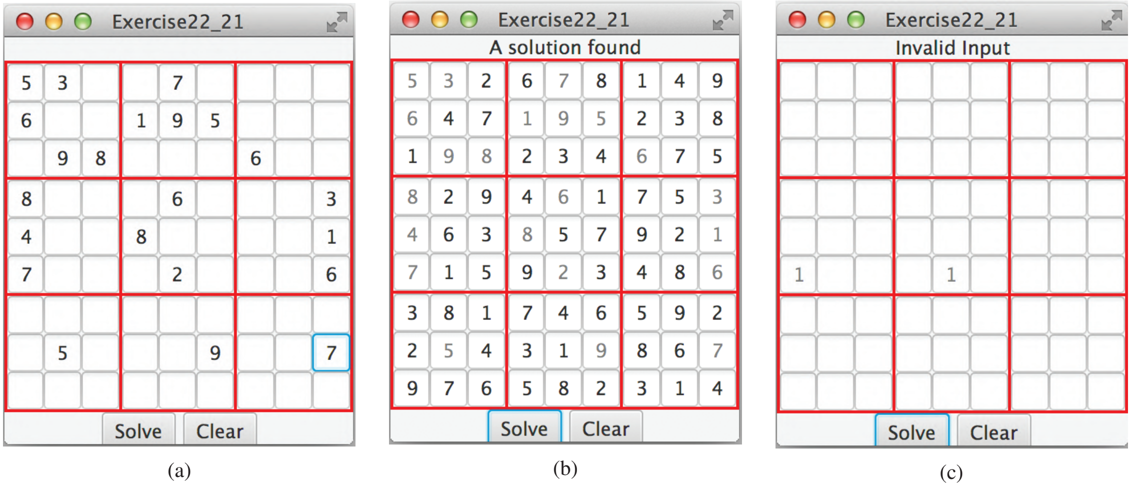
0	0	0	0	0	1	0	0	1	1
1	1	0	0	1	0	0	1	1	1
1	1	0	0	1	1	1	0	0	0
0	1	0	0	1	1	1	1	0	0
1	0	1	1	1	1	1	0	0	1
0	0	1	0	0	0	1	0	1	0
0	1	0	1	1	1	1	0	1	1
1	0	0	1	0	1	1	0	0	1
0	1	1	1	1	0	1	1	1	0
0	1	1	1	1	1	0	1	1	0

FIGURE 22.20 The program finds the largest block of 1s. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

*****22.21** (*Game: Sudoku*) The complete solution for the Sudoku problem is given in Supplement VI.C. Write a program that lets the user enter the input from the text fields, as shown in Figure 22.21a. Clicking the *Solve* button displays the result, as shown in Figures 22.21b and c.

*****22.22** (*Game: recursive Sudoku*) Write a recursive solution for the Sudoku problem.

*****22.23** (*Game: multiple Eight Queens solution*) Write a program to display all possible solutions for the Eight Queens puzzle in a scroll pane, as shown in Figure 22.22. For each solution, put a label to denote the solution number. (*Hint*: Place all solution panes into an `HBox` and place this one pane into a `ScrollPane`. If you run into a `StackOverflowError`, run the program using **java –Xss200m Exercise22_23** from the command window.)



(a)

5	3			7				
6			1	9	5			
	9	8				6		
8				6				3
4			8					1
7				2				6
	5				9			7

(b)

A solution found

5	3	2	6	7	8	1	4	9
6	4	7	1	9	5	2	3	8
1	9	8	2	3	4	6	7	5
8	2	9	4	6	1	7	5	3
4	6	3	8	5	7	9	2	1
7	1	5	9	2	3	4	8	6
3	8	1	7	4	6	5	9	2
2	5	4	3	1	9	8	6	7
9	7	6	5	8	2	3	1	4

(c)

Invalid Input

1				1				

FIGURE 22.21 The program solves the Sudoku problem. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

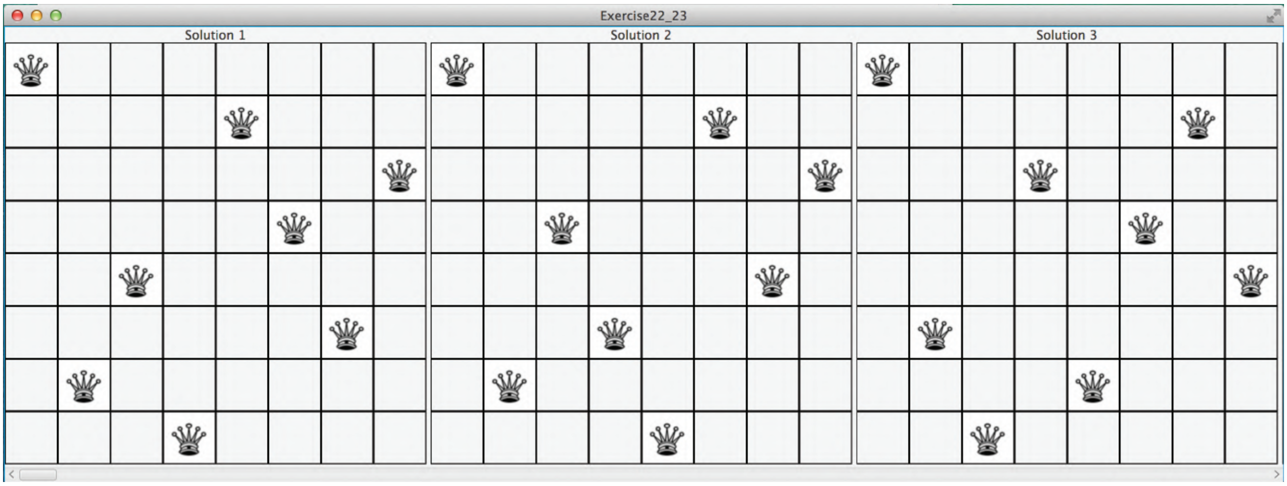
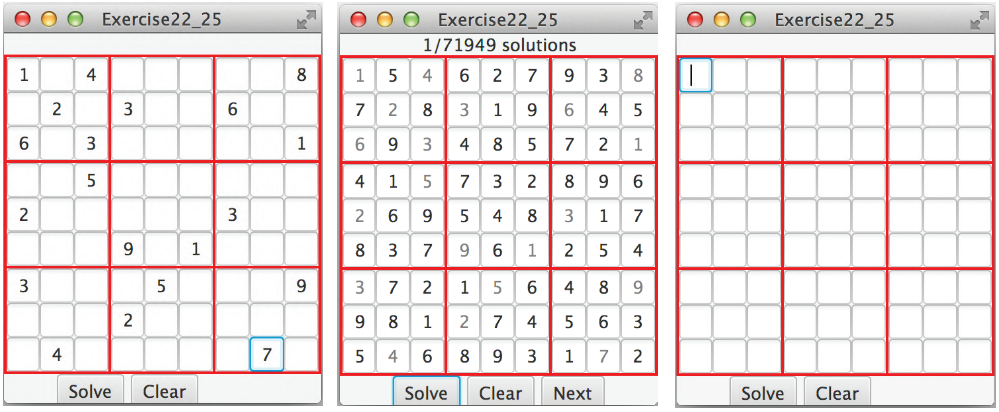

Exercise22_23
Solution 1
Solution 2
Solution 3

FIGURE 22.22 All solutions are placed in a scroll pane. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***22.24** (*Find the smallest number*) Write a method that uses the divide-and-conquer approach to find the smallest number in a list.

*****22.25** (*Game: Sudoku*) Revise Programming Exercise 22.21 to display all solutions for the Sudoku game, as shown in Figure 22.23a. When you click the *Solve* button, the program stores all solutions in an `ArrayList`. Each element in the list is a two-dimensional 9-by-9 grid. If the program has multiple solutions, the *Next* button appears as shown in Figure 22.23b. You can click the *Next* button to display the next solution and also add a label to show the solution count. When the *Clear* button is clicked, the cells are cleared and the *Next* button is hidden as shown in Figure 22.23c.


Exercise22_25
Solve
Clear
Exercise22_25
Solve
Clear
Next
Exercise22_25
Solve
Clear

1		4						8
	2		3			6		
6		3						1
		5						
2						3		
			9		1			
3				5				9
			2					
	4						7	

1/71949 solutions

1	5	4	6	2	7	9	3	8
7	2	8	3	1	9	6	4	5
6	9	3	4	8	5	7	2	1
4	1	5	7	3	2	8	9	6
2	6	9	5	4	8	3	1	7
8	3	7	9	6	1	2	5	4
3	7	2	1	5	6	4	8	9
9	8	1	2	7	4	5	6	3
5	4	6	8	9	3	1	7	2

(a) (b) (c)

FIGURE 22.23 The program can display multiple Sudoku solutions. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***22.26** (*Bin packing with smallest object first*) The bin packing problem is to pack the objects of various weights into containers. Assume each container can hold a maximum of 10 pounds. The program uses an algorithm that places an object with the *smallest weight* into the first bin in which it would fit. Your program should prompt the user to enter the total number of objects and the weight of each object. The program displays the total number of containers needed to pack the objects, and the contents of each container. Here is a sample run of the program:

```
Enter the number of objects: 6
Enter the weights of the objects: 7 5 2 3 5 8 ↵Enter
Container 1 contains objects with weight 2 3 5
Container 2 contains objects with weight 5
Container 3 contains objects with weight 7
Container 4 contains objects with weight 8
```

Does this program produce an optimal solution, that is, finding the minimum number of containers to pack the objects?

****22.27** (*Optimal bin packing*) Rewrite the preceding program so that it finds an optimal solution that packs all objects using the smallest number of containers. Here is a sample run of the program:

```
Enter the number of objects: 6 ↵Enter
Enter the weights of the objects: 7 5 2 3 5 8 ↵Enter
Container 1 contains objects with weight 7 3
Container 2 contains objects with weight 5 5
Container 3 contains objects with weight 2 8
The optimal number of bins is 3
```

What is the time complexity of your program?

CHAPTER 23

SORTING

Objectives

- To study and analyze time complexity of various sorting algorithms (§§23.2–23.7).
- To design, implement, and analyze insertion sort (§23.2).
- To design, implement, and analyze bubble sort (§23.3).
- To design, implement, and analyze merge sort (§23.4).
- To design, implement, and analyze quick sort (§23.5).
- To design and implement a binary heap (§23.6).
- To design, implement, and analyze heap sort (§23.6).
- To design, implement, and analyze bucket and radix sorts (§23.7).
- To design, implement, and analyze external sort for files that have a large amount of data (§23.8).

Key Point

23.1 Introduction

Sorting algorithms are good examples for studying algorithm design and analysis.

When president Barack Obama visited Google in 2007, the Google CEO Eric Schmidt asked Obama the most efficient way to sort a million 32-bit integers (www.youtube.com/watch?v=k4RRi_ntQc8). Obama answered that the bubble sort would be the wrong way to go. Was he right? We will examine different sorting algorithms in this chapter and see if he was correct.

why study sorting?

Sorting is a classic subject in computer science. There are three reasons to study sorting algorithms.

- First, sorting algorithms illustrate many creative approaches to problem solving, and these approaches can be applied to solve other problems.
- Second, sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, methods, and arrays.
- Third, sorting algorithms are excellent examples to demonstrate algorithm performance.

what data to sort?

The data to be sorted might be integers, doubles, characters, or objects. Section 7.11, Sorting Arrays, presented selection sort. The selection-sort algorithm was extended to sort an array of objects in Section 19.5, Case Study: Sorting an Array of Objects. The Java API contains several overloaded sort methods for sorting primitive-type values and objects in the `java.util.Arrays` and `java.util.Collections` classes. For simplicity, this chapter assumes

1. data to be sorted are integers,
2. data are stored in an array, and
3. data are sorted in ascending order.

The programs can be easily modified to sort other types of data, to sort in descending order, or to sort data in an `ArrayList` or a `LinkedList`.

There are many algorithms for sorting. You have already learned selection sort. This chapter introduces insertion sort, bubble sort, merge sort, quick sort, bucket sort, radix sort, and external sort.

23.2 Insertion Sort

Key Point

The insertion-sort algorithm sorts a list of values by repeatedly inserting a new element into a sorted sublist until the whole list is sorted.

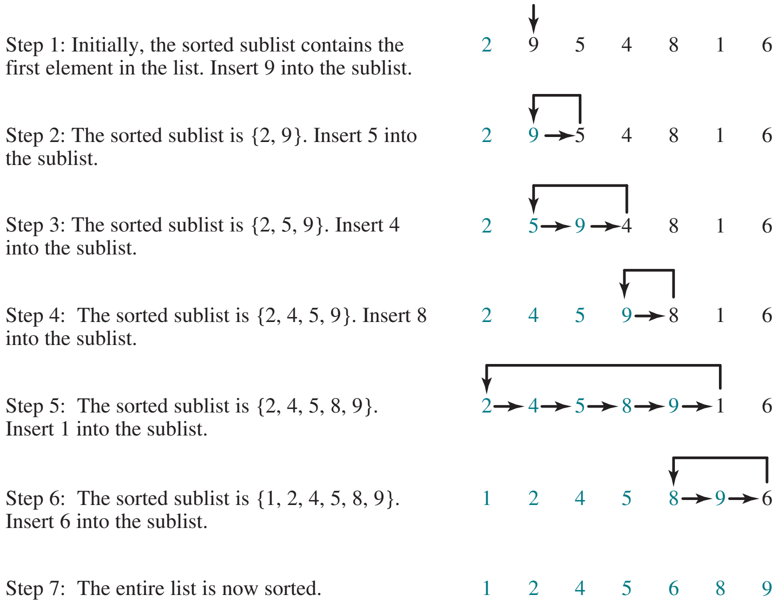
insertion-sort animation on Companion Website

Figure 23.1 shows how to sort a list {`2`, `9`, `5`, `4`, `8`, `1`, `6`} using insertion sort. For an interactive demo on how insertion sort works, go to liveexample.pearsoncmg.com/dsanimation/InsertionSortNeweBook.html.

The algorithm can be described as follows:

```
for (int i = 1; i < list.length; i++) {
  insert list[i] into a sorted sublist list[0..i-1] so that
  list[0..i] is sorted.
}
```

To insert `list[i]` into `list[0..i-1]`, save `list[i]` into a temporary variable, say `currentElement`. Move `list[i-1]` to `list[i]` if `list[i-1] > currentElement`, move `list[i-2]` to `list[i-1]` if `list[i-2] > currentElement`, and so on, until `list[i-k] <= currentElement` or `k > i` (we pass the first element of the sorted list). Assign `currentElement` to `list[i-k+1]`. For example, to insert `4` into {`2`, `5`, `9`} in Step 4 in Figure 23.2, move `list[2]` (`9`) to `list[3]` since `9 > 4` and move `list[1]` (`5`) to `list[2]` since `5 > 4`. Finally, move `currentElement` (`4`) to `list[1]`.



Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

Step 2: The sorted sublist is {2, 9}. Insert 5 into the sublist.

Step 3: The sorted sublist is {2, 5, 9}. Insert 4 into the sublist.

Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 into the sublist.

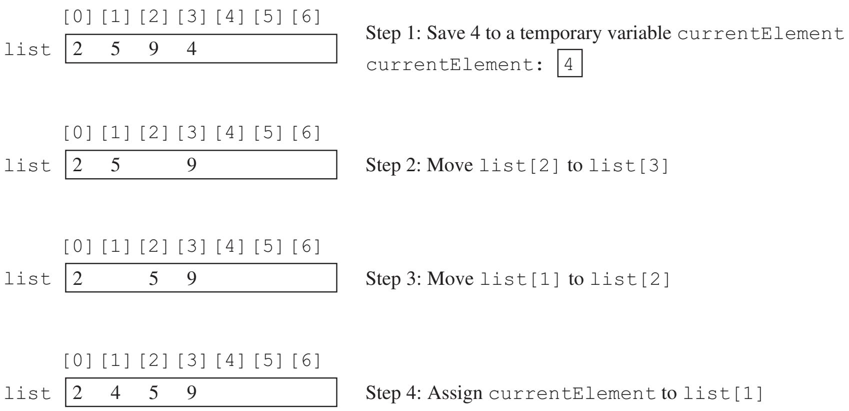
Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 into the sublist.

Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.

Step 7: The entire list is now sorted.

2 9 5 4 8 1 6
2 9 5 4 8 1 6
2 5 9 4 8 1 6
2 4 5 9 8 1 6
2 4 5 8 9 1 6
1 2 4 5 8 9 6
1 2 4 5 6 8 9

FIGURE 23.1 Insertion sort repeatedly inserts a new element into a sorted sublist.



Step 1: Save 4 to a temporary variable `currentElement`

Step 2: Move `list[2]` to `list[3]`

Step 3: Move `list[1]` to `list[2]`

Step 4: Assign `currentElement` to `list[1]`

[0] [1] [2] [3] [4] [5] [6]
list
2 5 9 4
currentElement:
4
[0] [1] [2] [3] [4] [5] [6]
list
2 5 9
[0] [1] [2] [3] [4] [5] [6]
list
2 5 9
[0] [1] [2] [3] [4] [5] [6]
list
2 4 5 9

FIGURE 23.2 A new element is inserted into a sorted sublist.

The algorithm can be expanded and implemented as in Listing 23.1.

LISTING 23.1 InsertionSort.java

```java
public class InsertionSort {
  /** The method for sorting the numbers */
  public static void insertionSort(int[] list) {
    for (int i = 1; i < list.length; i++) {
      /** Insert list[i] into a sorted sublist list[0..i-1] so that
          list[0..i] is sorted. */
      int currentElement = list[i];
      int k;
      for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
        list[k + 1] = list[k];
      }

      // Insert the current element into list[k + 1]
      list[k + 1] = currentElement;
    }
  }
}
```

shift

insert

The `insertionSort(int[] list)` method sorts an array of `int` elements. The method is implemented with a nested `for` loop. The outer loop (with the loop control variable `i`) (line 4) is iterated in order to obtain a sorted sublist, which ranges from `list[0]` to `list[i]`. The inner loop (with the loop control variable `k`) inserts `list[i]` into the sublist from `list[0]` to `list[i-1]`.

To better understand this method, trace it with the following statements:

```
int[] list = {1, 9, 4, 6, 5, -4};
InsertionSort.insertionSort(list);
```

insertion-sort time complexity

The insertion-sort algorithm presented here sorts a list of elements by repeatedly inserting a new element into a sorted partial array until the whole array is sorted. At the *k*th iteration, to insert an element into an array of size *k*, it may take *k* comparisons to find the insertion position and *k* moves to insert the element. Let $T(n)$ denote the complexity for insertion sort, and c denote the total number of other operations such as assignments and additional comparisons in each iteration. Thus,

$$\begin{aligned} T(n) &= (2 + c) + (2 \times 2 + c) + \cdots + (2 \times (n - 1) + c) \\ &= 2(1 + 2 + \cdots + n - 1) + c(n - 1) \\ &= 2\frac{(n - 1)n}{2} + cn - c = n^2 - n + cn - c \\ &= O(n^2) \end{aligned}$$

Therefore, the complexity of the insertion-sort algorithm is $O(n^2)$. Hence, the selection and insertion sorts are of the same time complexity.

Check Point

23.2.1 Describe how an insertion sort works. What is the time complexity for an insertion sort?

23.2.2 Use Figure 23.1 as an example to show how to apply an insertion sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.2.3 If a list is already sorted, how many comparisons will the `insertionSort` method perform?

23.3 Bubble Sort

Key Point

A bubble sort sorts the array in multiple passes. Each pass successively swaps the neighboring elements if the elements are not in order.

bubble sort

The bubble-sort algorithm makes several passes through the array. On each pass, successive neighboring pairs are compared. If a pair is in decreasing order, its values are swapped; otherwise, the values remain unchanged. The technique is called a *bubble sort* or *sinking sort* because the smaller values gradually “bubble” their way to the top and the larger values sink to the bottom. After the first pass, the last element becomes the largest in the array. After the second pass, the second-to-last element becomes the second largest in the array. This process is continued until all elements are sorted.

bubble-sort illustration

Figure 23.3a shows the first pass of a bubble sort on an array of six elements (2 9 5 4 8 1). Compare the elements in the first pair (2 and 9) and no swap is needed because they are already in order. Compare the elements in the second pair (9 and 5) and swap 9 with 5 because 9 is greater than 5. Compare the elements in the third pair (9 and 4) and swap 9 with 4. Compare the elements in the fourth pair (9 and 8) and swap 9 with 8. Compare the elements in the fifth pair (9 and 1) and swap 9 with 1. The pairs being compared are highlighted and the numbers already sorted are italicized in Figure 23.3. For an interactive demo on how bubble sort works, go to liveexample.pearsoncmg.com/dsanimation/BubbleSortNeweBook.html.

bubble sort animation on the Companion Website

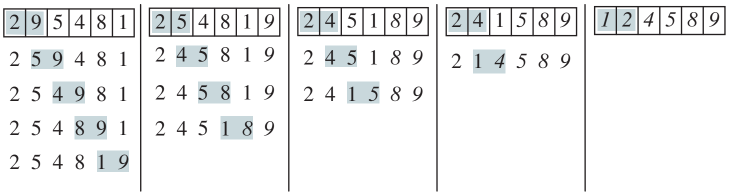

(a) 1st pass (b) 2nd pass (c) 3rd pass (d) 4th pass (e) 5th pass

FIGURE 23.3 Each pass compares and orders the pairs of elements sequentially.

The first pass places the largest number (9) as the last in the array. In the second pass, as shown in Figure 23.3b, you compare and order pairs of elements sequentially. There is no need to consider the last pair because the last element in the array is already the largest. In the third pass, as shown in Figure 23.3c, you compare and order pairs of elements sequentially except the last two elements because they are already in order. Thus, in the *k*th pass, you don't need to consider the last $k - 1$ elements because they are already ordered.

The algorithm for a bubble sort is described in Listing 23.2.

algorithm

LISTING 23.2 Bubble-Sort Algorithm

```
for (int k = 1; k < list.length; k++) {
  // Perform the kth pass
  for (int i = 0; i < list.length - k; i++) {
    if (list[i] > list[i + 1])
      swap list[i] with list[i + 1];
  }
}
```

Note if no swap takes place in a pass, there is no need to perform the next pass because all the elements are already sorted. You can use this property to improve the algorithm in Listing 23.2, as in Listing 23.3.

LISTING 23.3 Improved Bubble-Sort Algorithm

```
boolean needNextPass = true;
for (int k = 1; k < list.length && needNextPass; k++) {
  // Array may be sorted and next pass not needed
  needNextPass = false;
  // Perform the kth pass
  for (int i = 0; i < list.length – k; i++) {
    if (list[i] > list[i + 1]) {
      swap list[i] with list[i + 1];
      needNextPass = true; // Next pass still needed
    }
  }
}
```

The algorithm can be implemented in Listing 23.4.

LISTING 23.4 BubbleSort.java

```
public class BubbleSort {
  /** Bubble sort method */
  public static void bubbleSort(int[] list) {
    boolean needNextPass = true;

    for (int k = 1; k < list.length && needNextPass; k++) {
      // Array may be sorted and next pass not needed
      needNextPass = false;
      for (int i = 0; i < list.length - k; i++) {
```

perform one pass

```
        if (list[i] > list[i + 1]) {
          // Swap list[i] with list[i + 1]
          int temp = list[i];
          list[i] = list[i + 1];
          list[i + 1] = temp;

          needNextPass = true; // Next pass still needed
        }
      }
    }
  }

  /** A test method */
  public static void main(String[] args) {
    int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
    bubbleSort(list);
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }

}
```

```
-2 1 2 2 3 3 5 6 12 14
```

bubble-sort time complexity

In the best case, the bubble-sort algorithm needs just the first pass to find that the array is already sorted—no next pass is needed. Since the number of comparisons is $n - 1$ in the first pass, the best-case time for a bubble sort is $O(n)$.

In the worst case, the bubble-sort algorithm requires $n - 1$ passes. The first pass makes $n - 1$ comparisons, the second pass makes $n - 2$ comparisons, and so on; the last pass makes 1 comparison. Thus, the total number of comparisons is as follows:

$$(n - 1) + (n - 2) + \cdots + 2 + 1$$

$$= \frac{(n - 1)n}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

Therefore, the worst-case time for a bubble sort is $O(n^2)$.

Check Point

23.3.1 Describe how a bubble sort works. What is the time complexity for a bubble sort?

23.3.2 Use Figure 23.3 as an example to show how to apply a bubble sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.3.3 If a list is already sorted, how many comparisons will the `bubbleSort` method perform?

23.4 Merge Sort

Key Point

The merge-sort algorithm can be described recursively as follows: The algorithm divides the array into two halves and applies a merge sort on each half recursively. After the two halves are sorted, the algorithm then merges them.

merge sort

The algorithm for a *merge sort* is given in Listing 23.5.

LISTING 23.5 Merge-Sort Algorithm

base condition
sort first half
sort second half
merge two halves

```
public static void mergeSort(int[] list) {
  if (list.length > 1) {
    mergeSort(list[0 ... list.length / 2]);
    mergeSort(list[list.length / 2 + 1 ... list.length]);
    merge list[0 ... list.length / 2] with
```

```
      list[list.length / 2 + 1 ... list.length];
  }
}
```

merge-sort illustration

Figure 23.4 illustrates a merge sort of an array of eight elements (2 9 5 4 8 1 6 7). The original array is split into (2 9 5 4) and (8 1 6 7). Apply a merge sort on these two subarrays recursively to split (2 9 5 4) into (2 9) and (5 4) and (8 1 6 7) into (8 1) and (6 7). This process continues until the subarray contains only one element. For example, array (2 9) is split into the subarrays (2) and (9). Since array (2) contains a single element, it cannot be further split. Now merge (2) with (9) into a new sorted array (2 9) and (5) with (4) into a new sorted array (4 5). Merge (2 9) with (4 5) into a new sorted array (2 4 5 9) and finally merge (2 4 5 9) with (1 6 7 8) into a new sorted array (1 2 4 5 6 7 8 9).

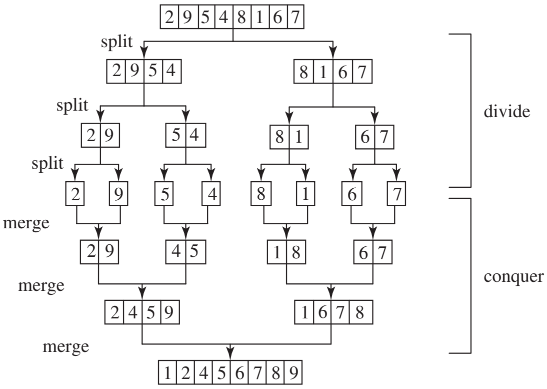


FIGURE 23.4 Merge sort employs a divide-and-conquer approach to sort the array.

The recursive call continues dividing the array into subarrays until each subarray contains only one element. The algorithm then merges these small subarrays into larger sorted subarrays until one sorted array results.

The merge-sort algorithm is implemented in Listing 23.6.

LISTING 23.6 MergeSort.java

```
public class MergeSort {
  /** The method for sorting the numbers */
  public static void mergeSort(int[] list) {
    if (list.length > 1) {
      // Merge sort the first half
      int[] firstHalf = new int[list.length / 2];
      System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
      mergeSort(firstHalf);

      // Merge sort the second half
      int secondHalfLength = list.length - list.length / 2;
      int[] secondHalf = new int[secondHalfLength];
      System.arraycopy(list, list.length / 2,
        secondHalf, 0, secondHalfLength);
      mergeSort(secondHalf);

      // Merge firstHalf with secondHalf into list
      merge(firstHalf, secondHalf, list);
    }
  }

  /** Merge two sorted lists */
  public static void merge(int[] list1, int[] list2, int[] temp) {
    int current1 = 0; // Current index in list1
```

base case

sort first half

sort second half

merge two halves

`list1` to `temp`

`list2` to `temp`

rest of `list1` to `temp`

rest of `list2` to `temp`

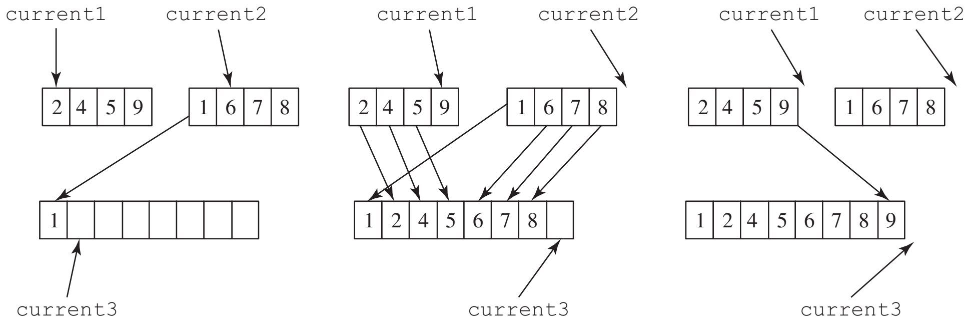
```
25       int current2 = 0; // Current index in list2
26       int current3 = 0; // Current index in temp
27
28       while (current1 < list1.length && current2 < list2.length) {
29         if (list1[current1] < list2[current2])
30           temp[current3++] = list1[current1++];
31         else
32           temp[current3++] = list2[current2++];
33       }
34
35       while (current1 < list1.length)
36         temp[current3++] = list1[current1++];
37
38       while (current2 < list2.length)
39         temp[current3++] = list2[current2++];
40     }
41
42     /** A test method */
43     public static void main(String[] args) {
44       int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
45       mergeSort(list);
46       for (int i = 0; i < list.length; i++)
47         System.out.print(list[i] + " ");
48     }
49   }
```

The `mergeSort` method (lines 3–20) creates a new array `firstHalf`, which is a copy of the first half of `list` (line 7). The algorithm invokes `mergeSort` recursively on `firstHalf` (line 8). The length of the `firstHalf` is `list.length / 2` and the length of the `secondHalf` is `list.length - list.length / 2`. The new array `secondHalf` was created to contain the second part of the original array `list`. The algorithm invokes `mergeSort` recursively on `secondHalf` (line 15). After `firstHalf` and `secondHalf` are sorted, they are merged to `list` (line 18). Thus, array `list` is now sorted.

The `merge` method (lines 23–40) merges two sorted arrays `list1` and `list2` into array `temp`. `current1` and `current2` point to the current element to be considered in `list1` and `list2` (lines 24–26). The method repeatedly compares the current elements from `list1` and `list2` and moves the smaller one to `temp`. `current1` is increased by `1` (line 30) if the smaller one is in `list1`, and `current2` is increased by `1` (line 32) if the smaller one is in `list2`. Finally, all the elements in one of the lists are moved to `temp`. If there are still unmoved elements in `list1`, copy them to `temp` (lines 35–36). If there are still unmoved elements in `list2`, copy them to `temp` (lines 38–39).

merge animation on Companion Website

Figure 23.5 illustrates how to merge the two arrays `list1` (2 4 5 9) and `list2` (1 6 7 8). Initially, the current elements to be considered in the arrays are 2 and 1. Compare



(a) After moving `1` to `temp`

(b) After moving all the elements in `list2` to `temp`

(c) After moving `9` to `temp`

FIGURE 23.5 Two sorted arrays are merged into one sorted array.

them and move the smaller element 1 to `temp`, as shown in Figure 23.5a. `current2` and `current3` are increased by 1. Continue to compare the current elements in the two arrays and move the smaller one to `temp` until one of the arrays is completely moved. As shown in Figure 23.5b, all the elements in `list2` are moved to `temp` and `current1` points to element 9 in `list1`. Copy 9 to `temp`, as shown in Figure 23.5c. For an interactive demo on how merge works, go to liveexample.pearsoncmg.com/dsanimation/MergeSortNeweBook.html.

The `mergeSort` method creates two temporary arrays (lines 6 and 12) during the dividing process, copies the first half and the second half of the array into the temporary arrays (lines 7 and 13), sorts the temporary arrays (lines 8 and 15), then merges them into the original array (line 18), as shown in Figure 23.6a. You can rewrite the code to recursively sort the first half of the array and the second half of the array without creating new temporary arrays, then merge the two arrays into a temporary array and copy its contents to the original array, as shown in Figure 23.6b. This is left for you to do in Programming Exercise 23.20.

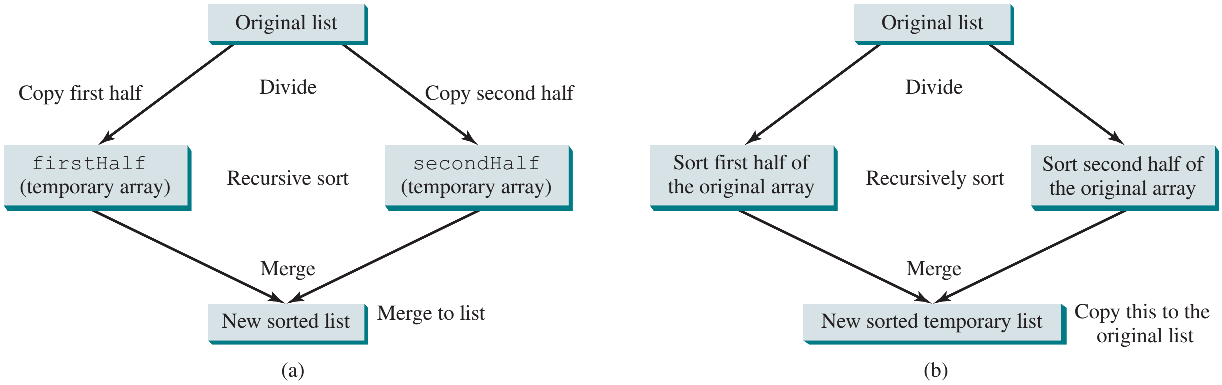

Original list
Copy first half
Divide
Copy second half
firstHalf (temporary array)
Recursive sort
secondHalf (temporary array)
Merge
New sorted list
Merge to list
(a)
Original list
Divide
Sort first half of the original array
Recursively sort
Sort second half of the original array
Merge
New sorted temporary list
Copy this to the original list
(b)

FIGURE 23.6 Temporary arrays are created to support a merge sort.

Note

A merge sort can be implemented efficiently using parallel processing. See Section 32.16, Parallel Programming, for a parallel implementation of a merge sort.

merge-sort time complexity

Let us analyze the running time of merge sort now. Let $T(n)$ denote the time required for sorting an array of n elements using a merge sort. Without loss of generality, assume n is a power of 2. The merge-sort algorithm splits the array into two subarrays, sorts the subarrays using the same algorithm recursively, then merges the subarrays. Therefore,

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + mergetime$$

The first $T\left(\frac{n}{2}\right)$ is the time for sorting the first half of the array and the second $T\left(\frac{n}{2}\right)$ is the time for sorting the second half. To merge two subarrays, it takes at most $n - 1$ comparisons to compare the elements from the two subarrays, and n moves to move elements to the temporary array. Thus, the total time is $2n - 1$. Therefore,

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2n - 1 = O(n \log n)$$

$O(n \log n)$ merge sort

The complexity of a merge sort is $O(n \log n)$. This algorithm is better than selection sort, insertion sort, and bubble sort because the time complexity of these algorithms is $O(n^2)$. The `sort` method in the `java.util.Arrays` class is implemented using a variation of the merge-sort algorithm.

Check Point

23.4.1 Describe how a merge sort works. What is the time complexity for a merge sort?

23.4.2 Use Figure 23.4 as an example to show how to apply a merge sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.4.3 What is wrong if lines 6–15 in Listing 23.6, MergeSort.java, are replaced by the following code?

```
// Merge sort the first half
int[] firstHalf = new int[list.length /  2 + 1];
System.arraycopy(list, 0, firstHalf, 0, list.length / 2 + 1);
mergeSort(firstHalf);

// Merge sort the second half
int secondHalfLength = list.length - list.length / 2 - 1;
int[] secondHalf = new int[secondHalfLength];
System.arraycopy(list, list.length / 2 + 1,
  secondHalf, 0, secondHalfLength);
mergeSort(secondHalf);
```

23.5 Quick Sort

Key Point

quick sort

A **`quick sort`** *works as follows: The algorithm selects an element, called the pivot, in the array. It divides the array into two parts so all the elements in the first part are less than or equal to the pivot, and all the elements in the second part are greater than the pivot. The quick-sort algorithm is then recursively applied to the first part and then the second part.*

The quick-sort algorithm, developed by C.A.R. Hoare in 1962, is described in Listing 23.7.

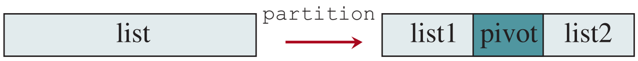
Listing 23.7 Quick-Sort Algorithm

base condition
select the pivot
partition the list
sort first part
sort second part

```
public static void quickSort(int[] list) {
  if (list.length > 1) {
    select a pivot;
    partition list into list1 and list2 such that
      all elements in list1 <= pivot and
      all elements in list2 > pivot;
    quickSort(list1);
    quickSort(list2);
  }
}
```

how to partition

Each partition places the pivot in the right place. It divides the list into two sublists as shown in the following figure.

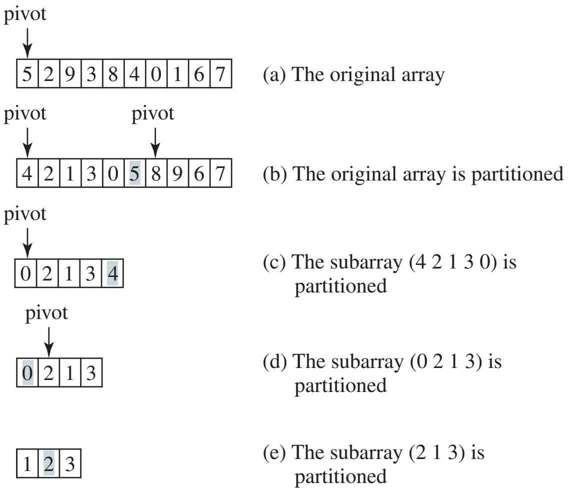


The selection of the pivot affects the performance of the algorithm. Ideally, the algorithm should choose the pivot that divides the two parts evenly. For simplicity, assume that the first element in the array is chosen as the pivot. (Programming Exercise 23.4 proposes an alternative strategy for selecting the pivot.)

quick-sort illustration

Figure 23.7 illustrates how to sort an array (5 2 9 3 8 4 0 1 6 7) using quick sort. Choose the first element, 5, as the pivot. The array is partitioned into two parts, as shown in

Figure 23.7b. The highlighted pivot is placed in the right place in the array. Apply quick sort on two subarrays (4 2 1 3 0) then (8 9 6 7). The pivot 4 partitions (4 2 1 3 0) into just one subarrays (0 2 1 3), as shown in Figure 23.7c. Apply quick sort on (0 2 1 3). The pivot 0 partitions it into just one subarrays (2 1 3), as shown in Figure 23.7d. Apply quick sort on (2 1 3). The pivot 2 partitions it into (1) and (3), as shown in Figure 23.7e. Apply quick sort on (1). Since the array contains just one element, no further partition is needed.



(a) The original array

(b) The original array is partitioned

(c) The subarray (4 2 1 3 0) is partitioned

(d) The subarray (0 2 1 3) is partitioned

(e) The subarray (2 1 3) is partitioned

FIGURE 23.7 The quick-sort algorithm is recursively applied to subarrays.

The quick-sort algorithm is implemented in Listing 23.8. There are two overloaded `quickSort` methods in the class. The first method (line 2) is used to sort an array. The second is a helper method (line 6) that sorts a subarray with a specified range.

LISTING 23.8 `QuickSort.java`

```
 1  public class QuickSort {
 2    public static void quickSort(int[] list) {
 3      quickSort(list, 0, list.length - 1);
 4    }
 5
 6    public static void quickSort(int[] list, int first, int last) {
 7      if (last > first) {
 8        int pivotIndex = partition(list, first, last);
 9        quickSort(list, first, pivotIndex -  1);
10        quickSort(list, pivotIndex + 1, last);
11      }
12    }
13
14    /** Partition the array list[first..last] */
15    public static int partition(int[] list, int first, int last) {
16      int pivot = list[first]; // Choose the first element as the pivot
17      int low = first + 1; // Index for forward search
18      int high = last; // Index for backward search
19
20      while (high > low) {
21        // Search forward from left
22        while (low <= high && list[low] <= pivot)
23          low++;
24
25        // Search backward from right
```

sort method

helper method

recursive call

forward

backward
swap
place pivot
pivot's new index
pivot's original index

```
26        while (low <= high && list[high] > pivot)
27          high--;
28
29        // Swap two elements in the list
30        if (high > low) {
31          int temp = list[high];
32          list[high] = list[low];
33          list[low] = temp;
34        }
35      }
36
37      while (high > first && list[high] >= pivot)
38        high--;
39
40      // Swap pivot with list[high]
41      if (pivot > list[high]) {
42        list[first] = list[high];
43        list[high] = pivot;
44        return high;
45      }
46      else {
47        return first;
48      }
49    }
50
51    /** A test method */
52    public static void main(String[] args) {
53      int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
54      quickSort(list);
55      for (int i = 0; i < list.length; i++)
56        System.out.print(list[i] + " ");
57    }
58  }
```

```
-2 1 2 2 3 3 5 6 12 14
```

The **`partition`** method (lines 15–49) partitions the array **`list[first..last]`** using the pivot. The first element in the partial array is chosen as the pivot (line 16). Initially, **`low`** points to the second element in the subarrays (line 17) and **`high`** points to the last element in the subarrays (line 18).

Starting from the left, the method searches forward in the array for the first element that is greater than the pivot (lines 22–23), then searches from the right backward for the first element in the array that is less than or equal to the pivot (lines 26–27). It then swaps these two elements and repeats the same search and swap operations until all the elements are searched in a **`while`** loop (lines 20–35).

The method returns the new index for the pivot that divides the subarrays into two parts if the pivot has been moved (line 44). Otherwise, it returns the original index for the pivot (line 47).

partition illustration

partition animation on Companion Website

Figure 23.8 illustrates how to partition an array (5 2 9 3 8 4 0 1 6 7). Choose the first element, 5, as the pivot. Initially, **`low`** is the index that points to element 2 and **`high`** points to element 7, as shown in Figure 23.8a. Advance index **`low`** forward to search for the first element (9) that is greater than the pivot, and move index **`high`** backward to search for the first element (1) that is less than or equal to the pivot, as shown in Figure 23.8b. Swap 9 with 1, as shown in Figure 23.8c. Continue the search and move **`low`** to point to element 8 and **`high`** to point to element 0, as shown in Figure 23.8d. Swap element 8 with 0, as shown in Figure 23.8e. Continue to move **`low`** until it passes **`high`**, as shown in Figure 23.8f. Now all the elements are examined. Swap the pivot with element 4 at index `high`. The final partition is shown in Figure 23.8g. The index of the pivot is returned when the method is finished. For an interactive demo on how partition works, go to liveexample.pearsoncmg.com/dsanimation/QuickSortNeweBook.html.

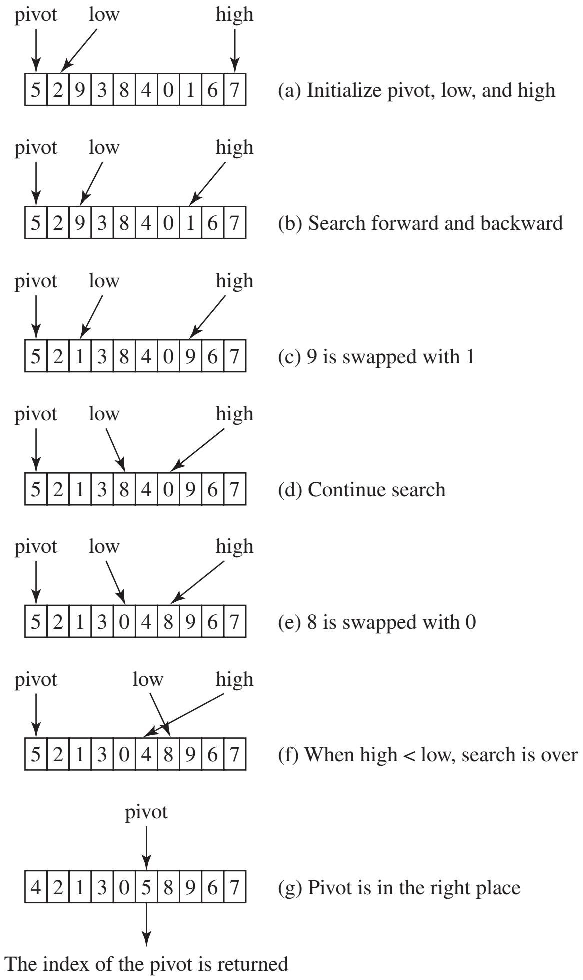

pivot
low
high
5 2 9 3 8 4 0 1 6 7
(a) Initialize pivot, low, and high
pivot
low
high
5 2 9 3 8 4 0 1 6 7
(b) Search forward and backward
pivot
low
high
5 2 1 3 8 4 0 9 6 7
(c) 9 is swapped with 1
pivot
low
high
5 2 1 3 8 4 0 9 6 7
(d) Continue search
pivot
low
high
5 2 1 3 0 4 8 9 6 7
(e) 8 is swapped with 0
pivot
low
high
5 2 1 3 0 4 8 9 6 7
(f) When high < low, search is over
pivot
4 2 1 3 0 5 8 9 6 7
(g) Pivot is in the right place
The index of the pivot is returned

FIGURE 23.8 The `partition` method returns the index of the pivot after it is put in the correct place.

To partition an array of n elements, it takes n comparisons and n moves in the worst case. Thus, the time required for partition is $O(n)$.

$O(n)$ partition time

$O(n^2)$ worst-case time

In the worst case, the pivot divides the array each time into one big subarray with the other array empty. The size of the big subarray is one less than the one before divided. The algorithm requires $(n - 1) + (n - 2) + \cdots + 2 + 1 = O(n^2)$ time.

$O(n \log n)$ best-case time

In the best case, the pivot divides the array each time into two parts of about the same size. Let $T(n)$ denote the time required for sorting an array of n elements using quick sort. Thus,

$$T(n) = \overbrace{T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right)}^{\text{recursive quick sort on two subarrays}} + \overbrace{n}^{\text{partition time}}.$$

Similar to the merge-sort analysis, $T(n) = O(n \log n)$.

O(*n* log*n*) average-case time

On the average, the pivot will not divide the array into two parts of the same size or one empty part each time. Statistically, the sizes of the two parts are very close. Therefore, the average time is *O*(*n* log*n*). The exact average-case analysis is beyond the scope of this book.

quick sort vs. merge sort

Both merge and quick sorts employ the divide-and-conquer approach. For merge sort, the bulk of the work is to merge two sublists, which takes place *after* the sublists are sorted. For quick sort, the bulk of the work is to partition the list into two sublists, which takes place *before* the sublists are sorted. Merge sort is more efficient than quick sort in the worst case, but the two are equally efficient in the average case. Merge sort requires a temporary array for sorting two subarrays. Quick sort does not need additional array space. Thus, quick sort is more space efficient than merge sort.

Check Point

23.5.1 Describe how quick sort works. What is the time complexity for a quick sort?

23.5.2 Why is quick sort more space efficient than merge sort?

23.5.3 Use Figure 23.7 as an example to show how to apply a quick sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.5.4 If lines 37–38 in the QuickSort program is removed, will it still work? Give a counter example to show that it will not work.

23.6 Heap Sort

Key Point

A heap sort uses a binary heap. It first adds all the elements to a heap and then removes the largest elements successively to obtain a sorted list.

heap sort
root
left subtree
right subtree
length
depth
leaf

Heap sorts use a binary heap, which is a complete binary tree. A binary tree is a hierarchical structure. It either is empty or it consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*. The *length* of a path is the number of the edges in the path. The *depth* of a node is the length of the path from the root to the node. A node is called a *leaf* if it does not have subtrees.

A *binary heap* is a binary tree with the following properties:

- Shape property: It is a complete binary tree.
- Heap property: Each node is greater than or equal to any of its children.

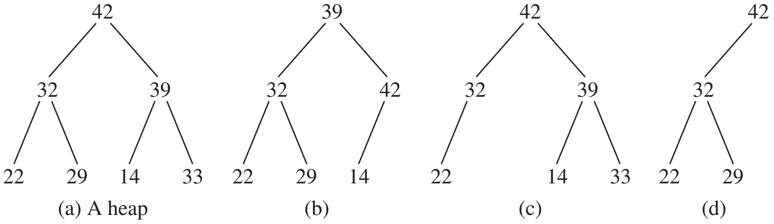


FIGURE 23.9 A binary heap is a special complete binary tree.

complete binary tree

A binary tree is *complete* if each of its levels is full, except that the last level may not be full and all the leaves on the last level are placed leftmost. For example, in Figure 23.9, the binary trees in (a) and (b) are complete, but the binary trees in (c) and (d) are not complete. Further, the binary tree in (a) is a heap, but the binary tree in (b) is not a heap because the root (39) is less than its right child (42).

heap

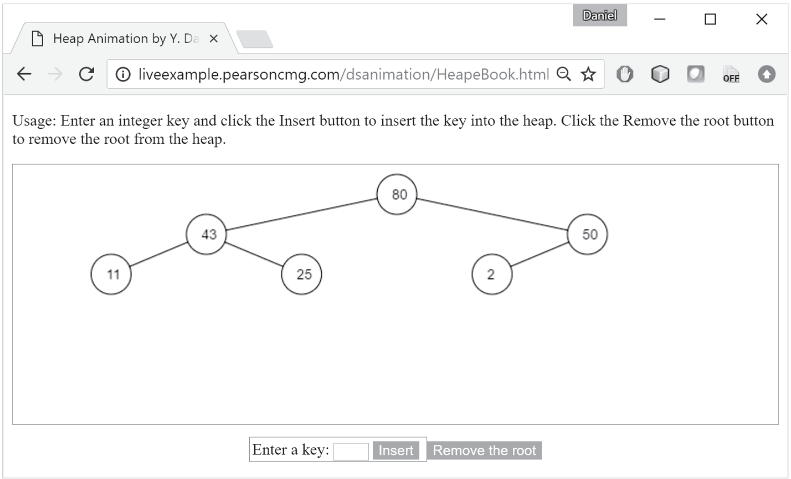
Note

Heap is a term with many meanings in computer science. In this chapter, heap means a binary heap.

Pedagogical Note

A heap can be implemented efficiently for inserting keys and deleting the root. For an interactive demo on how a heap works, go to liveexample.pearsoncmg.com/dsanimation/HeapeBook.html, as shown in Figure 23.10.

heap animation on Companion Website

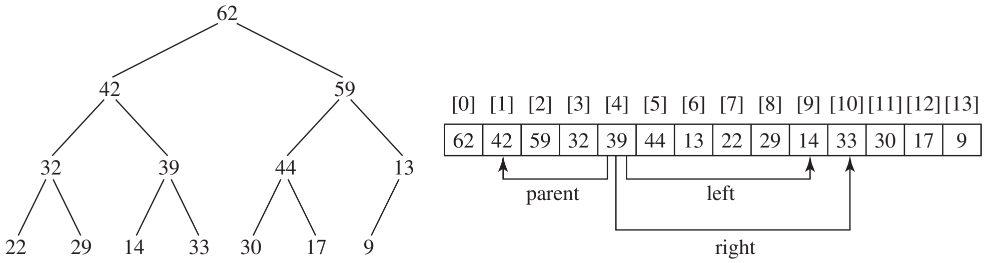

Daniel
Heap Animation by Y. Da ×
liveexample.pearsoncmg.com/dsanimation/HeapeBook.html
80
43
50
11
25
2
Enter a key:
Insert
Remove the root

Usage: Enter an integer key and click the Insert button to insert the key into the heap. Click the Remove the root button to remove the root from the heap.

FIGURE 23.10 The heap animation tool enables you to insert a key and delete the root visually. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

23.6.1 Storing a Heap

A heap can be stored in an `ArrayList` or an array if the heap size is known in advance. The heap in Figure 23.11a can be stored using the array in Figure 23.11b. The root is at position 0, and its two children are at positions 1 and 2. For a node at position i, its left child is at position $2i + 1$, its right child is at position $2i + 2$, and its parent is $(i - 1)/2$. For example, the node for element 39 is at position 4, so its left child (element 14) is at 9 $(2 \times 4 + 1)$, its right child (element 33) is at 10 $(2 \times 4 + 2)$, and its parent (element 42) is at 1 $((4 - 1)/2)$.


62
42
59
32
39
44
13
22
29
14
33
30
17
9
parent
left
right

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
62	42	59	32	39	44	13	22	29	14	33	30	17	9

(a) A heap

(b) A heap stored in an array

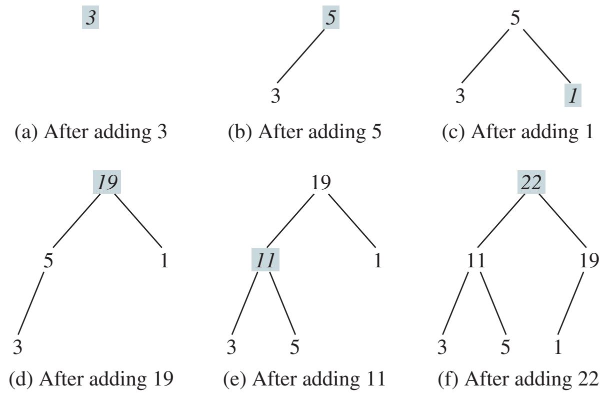
FIGURE 23.11 A binary heap can be implemented using an array.

23.6.2 Adding a New Node

To add a new node to the heap, first add it to the end of the heap then rebuild the tree as follows:

```
Let the last node be the current node;
while (the current node is greater than its parent) {
  Swap the current node with its parent;
  Now the current node is one level up;
}
```

Suppose a heap is initially empty. That heap is shown in Figure 23.12, after adding numbers 3, 5, 1, 19, 11, and 22 in this order.

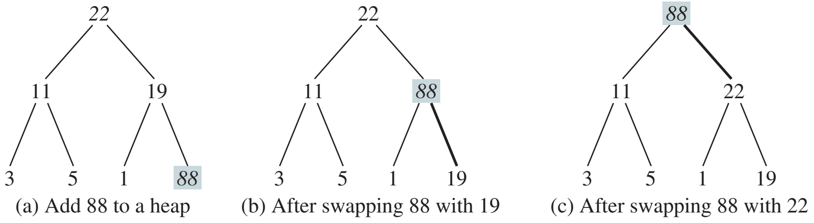


(a) After adding 3 (b) After adding 5 (c) After adding 1

(d) After adding 19 (e) After adding 11 (f) After adding 22

FIGURE 23.12 Elements 3, 5, 1, 19, 11, and 22 are inserted into the heap.

Now consider adding 88 into the heap. Place the new node 88 at the end of the tree, as shown in Figure 23.13a. Swap 88 with 19, as shown in Figure 23.13b. Swap 88 with 22, as shown in Figure 23.13c.



(a) Add 88 to a heap (b) After swapping 88 with 19 (c) After swapping 88 with 22

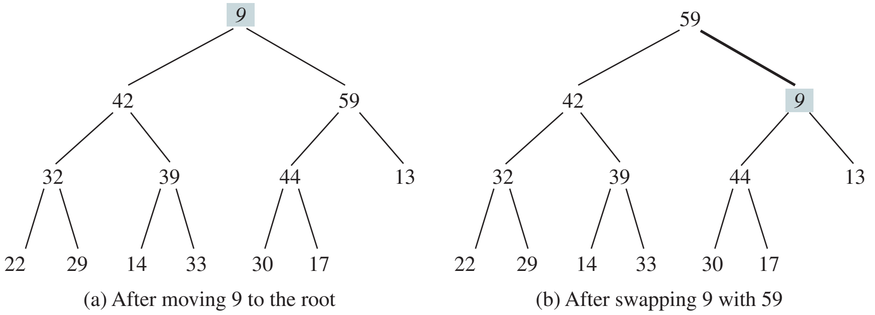
FIGURE 23.13 Rebuild the heap after adding a new node.

23.6.3 Removing the Root

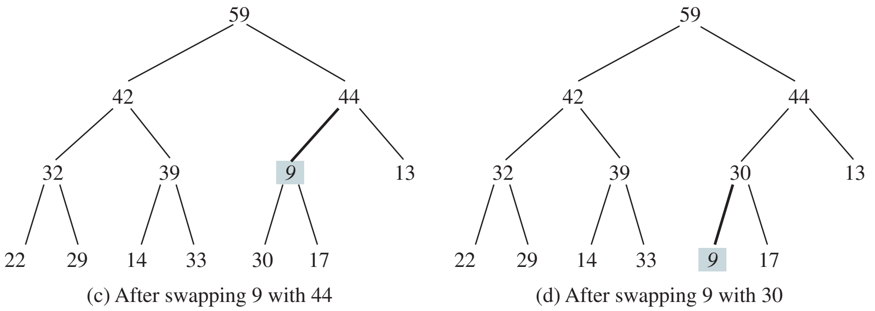
Often you need to remove the maximum element, which is the root in a heap. After the root is removed, the tree must be rebuilt to maintain the heap property. The algorithm for rebuilding the tree can be described as follows:

```
Move the last node to replace the root;
Let the root be the current node;
while (the current node has children and the current node is
       smaller than one of its children) {
  Swap the current node with the larger of its children;
  Now the current node is one level down;
}
```

Figure 23.14 shows the process of rebuilding a heap after the root 62 is removed from Figure 23.11a. Move the last node 9 to the root, as shown in Figure 23.14a. Swap 9 with 59, as shown in Figure 23.14b; swap 9 with 44, as shown in Figure 23.14c; and swap 9 with 30, as shown in Figure 23.14d.


9
42
59
32
39
44
13
22
29
14
33
30
17
59
42
9
32
39
44
13
22
29
14
33
30
17

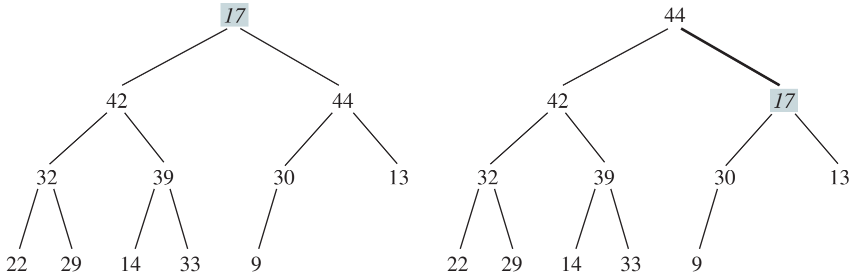
(a) After moving 9 to the root (b) After swapping 9 with 59


59
42
44
32
39
9
13
22
29
14
33
30
17
59
42
44
32
39
30
13
22
29
14
33
9
17

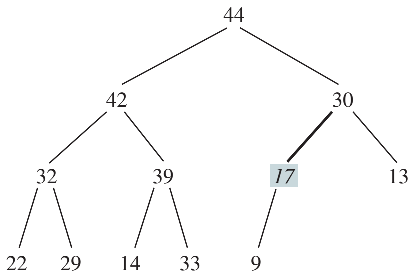
(c) After swapping 9 with 44 (d) After swapping 9 with 30

FIGURE 23.14 Rebuild the heap after the root 62 is removed.

Figure 23.15 shows the process of rebuilding a heap after the root 59 is removed from Figure 23.14d. Move the last node 17 to the root, as shown in Figure 23.15a. Swap 17 with 44, as shown in Figure 23.15b, then swap 17 with 30, as shown in Figure 23.15c.


17
42
44
32
39
30
13
22
29
14
33
9
44
42
17
32
39
30
13
22
29
14
33
9

(a) After moving 17 to the root (b) After swapping 17 with 44


44
42
30
32
39
17
13
22
29
14
33
9

(c) After swapping 17 with 30

FIGURE 23.15 Rebuild the heap after the root 59 is removed.

23.6.4 The **Heap** Class

Now you are ready to design and implement the **Heap** class. The class diagram is shown in Figure 23.16. Its implementation is given in Listing 23.9.

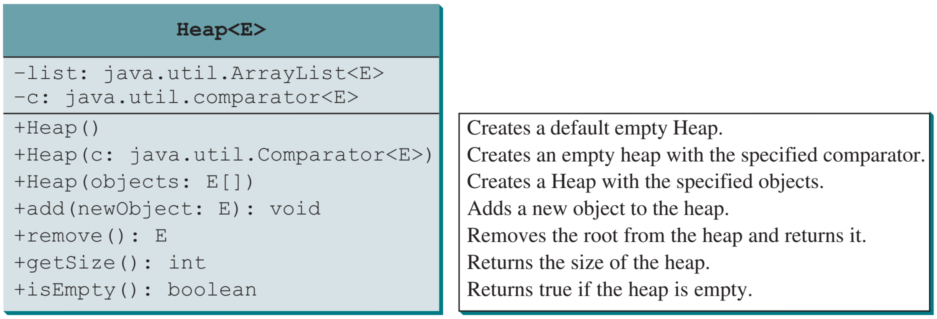

Heap<E>
-list: java.util.ArrayList<E>
-c: java.util.comparator<E>
+Heap()
+Heap(c: java.util.Comparator<E>)
+Heap(objects: E[])
+add(newObject: E): void
+remove(): E
+getSize(): int
+isEmpty(): boolean
Creates a default empty Heap.
Creates an empty heap with the specified comparator.
Creates a Heap with the specified objects.
Adds a new object to the heap.
Removes the root from the heap and returns it.
Returns the size of the heap.
Returns true if the heap is empty.

FIGURE 23.16 The **Heap** class provides operations for manipulating a heap.

LISTING 23.9 Heap.java

internal heap representation
comparator
no-arg constructor
create a comparator
internal heap representation
constructor
add a new object
append the object
swap with parent
heap now
remove the root
empty heap

```java
public class Heap<E> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<>();
  private java.util.Comparator<? super E> c;

  /** Create a default heap */
  public Heap() {
    this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
  }

  /** Create a heap with a specified comparator */
  public Heap(java.util.Comparator<E> c) {
    this.c = c;
  }

  /** Create a heap from an array of objects */
  public Heap(E[] objects) {
    this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
    for (int i = 0; i < objects.length; i++)
      add(objects[i]);
  }

  /** Add a new object into the heap */
  public void add(E newObject) {
    list.add(newObject); // Append to the heap
    int currentIndex = list.size() - 1; // The index of the last node

    while (currentIndex > 0) {
      int parentIndex = (currentIndex -  1) /  2;
      // Swap if the current object is greater than its parent
      if (c.compare(list.get(currentIndex),
          list.get(parentIndex)) > 0) {
        E temp = list.get(currentIndex);
        list.set(currentIndex, list.get(parentIndex));
        list.set(parentIndex, temp);
      }
      else
        break; // The tree is a heap now

      currentIndex = parentIndex;
    }
  }

  /** Remove the root from the heap */
  public E remove() {
    if (list.size() == 0)  return null;
```

root
new root
remove the last
adjust the tree
compare two children
swap with the larger child

```
46
47       E removedObject = list.get(0);
48       list.set(0, list.get(list.size() - 1));
49       list.remove(list.size() -  1);
50
51       int currentIndex = 0;
52       while (currentIndex < list.size()) {
53         int leftChildIndex = 2 * currentIndex + 1;
54         int rightChildIndex = 2 * currentIndex + 2;
55
56         // Find the maximum between two children
57         if (leftChildIndex >= list.size()) break; // The tree is a heap
58         int maxIndex = leftChildIndex;
59         if (rightChildIndex < list.size()) {
60           if (c.compare(list.get(maxIndex),
61               list.get(rightChildIndex)) < 0) {
62             maxIndex = rightChildIndex;
63           }
64         }
65
66         // Swap if the current node is less than the maximum
67         if (c.compare(list.get(currentIndex),
68                       list.get(maxIndex)) < 0) {
69           E temp = list.get(maxIndex);
70           list.set(maxIndex, list.get(currentIndex));
71           list.set(currentIndex, temp);
72           currentIndex = maxIndex;
73         }
74         else
75           break; // The tree is a heap
76       }
77
78       return removedObject;
79     }
80
81     /** Get the number of nodes in the tree */
82     public int getSize() {
83       return list.size();
84     }
85
86     /** Return true if heap is empty */
87     public boolean isEmpty() {
88       return list.size() == 0;
89     }
90   }
```

A heap is represented using an array list internally (line 2). You can change the array list to other data structures, but the **`Heap`** class contract will remain unchanged.

The elements in the heap can be compared using a comparator or using a natural order if no comparator is specified.

The no-arg constructor (line 6) creates an empty heap using a natural order for the elements as the comparator. The comparator is created using a lambda expression (line 7).

The **`add(E newObject)`** method (lines 23–41) appends the object to the tree then swaps the object with its parent if the object is greater than its parent. This process continues until the new object becomes the root or is not greater than its parent.

The **`remove()`** method (lines 44–79) removes and returns the root. To maintain the heap property, the method moves the last object to the root position and swaps it with its larger child if it is less than the larger child. This process continues until the last object becomes a leaf or is not less than its children.

23.6.5 Sorting Using the **Heap** Class

To sort an array using a heap, first create an object using the **`Heap`** class, add all the elements to the heap using the **`add`** method, and remove all the elements from the heap using the **`remove`** method. The elements are removed in descending order. Listing 23.10 gives a program for sorting an array using a heap.

LISTING 23.10 HeapSort.java

heapSort method default

heapSort method using comparator

create a Heap

add element

remove element

invoke sort method

```java
 1 import java.util.Comparator;
 2
 3 public class HeapSort {
 4   /** Heap sort method */
 5   public static <E> void heapSort(E[] list) {
 6     // Create a Heap of integers
 7     heapSort(list, (e1, e2) -> ((Comparable<E>)e1).compareTo(e2));
 8   }
 9
10   /** Heap sort method */
11   public static <E> void heapSort(E[] list, Comparator<E> c) {
12     // Create a Heap of integers
13     Heap<E> heap = new Heap<>(c);
14
15     // Add elements to the heap
16     for (int i = 0; i < list.length; i++)
17       heap.add(list[i]);
18
19     // Remove elements from the heap
20     for (int i = list.length - 1; i >= 0; i--)
21       list[i] = heap.remove();
22   }
23
24   /** A test method */
25   public static void main(String[] args) {
26     Integer[] list = {-44, -5, -3, 3, 3, 1, -4, 0, 1, 2, 4, 5, 53};
27     heapSort(list);
28     for (int i = 0; i < list.length; i++)
29       System.out.print(list[i] + " ");
30   }
31 }
```

```
-44 -5 -4 -3 0 1 1 2 3 3 4 5 53
```

Two **`heapSort`** methods are provided here. The **`heapSort(E[] list)`** method (line 5) sorts a list in a natural order using the **`Comparable`** interface. The **`heapSort(E[] list, Comparator<E> c)`** method (line 11) sorts a list using a specified comparator.

23.6.6 Heap Sort Time Complexity

height of a heap

Let us turn our attention to analyzing the time complexity for the heap sort. Let h denote the *height* for a heap of n elements. The height of a nonempty tree is the length of the path from the root node to its furthest leaf. The *height* of a tree that contains a single node is **`0`**. Conventionally, the height of an empty tree is **`-1`**. Since a heap is a complete binary tree, the first level has 1 (2^0) node, the second level has 2 (2^1) nodes, the kth level has 2^{k-1} nodes, the h level has 2^{h-1} nodes, and the last $(h + 1)$th level has at least 1 and at most 2^h nodes. Therefore,

$$1 + 2 + \cdots + 2^{h-1} < n \leq 1 + 2 + \cdots + 2^{h-1} + 2^h$$

That is,

heap sort vs. merge sort

$$2^h - 1 < n \leq 2^{h+1} - 1$$
$$2^h < n + 1 \leq 2^{h+1}$$
$$h < \log(n + 1) \leq h + 1$$

Thus, $h < \log(n + 1)$ and $h \geq \log(n + 1) - 1$. Therefore, $\log(n + 1) - 1 \leq h < \log(n + 1)$. Hence, the height of a heap is $O(\log n)$. More precisely, you can prove that $h = \lfloor \log n \rfloor$ for a non-empty tree.

$O(n \log n)$ worst-case time

Since the `add` method traces a path from a leaf to a root, it takes at most h steps to add a new element to the heap. Thus, the total time for constructing an initial heap is $O(n \log n)$ for an array of n elements. Since the `remove` method traces a path from a root to a leaf, it takes at most h steps to rebuild a heap after removing the root from the heap. Since the `remove` method is invoked n times, the total time for producing a sorted array from a heap is $O(n \log n)$.

Both merge and heap sorts require $O(n \log n)$ time. A merge sort requires a temporary array for merging two subarrays; a heap sort does not need additional array space. Therefore, a heap sort is more space efficient than a merge sort.

Check Point

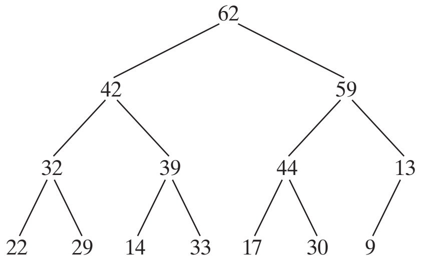
23.6.1 What is a complete binary tree? What is a heap? Describe how to remove the root from a heap and how to add a new object to a heap.

23.6.2 Add the elements `4`, `5`, `1`, `2`, `9`, and `3` into a heap in this order. Draw the diagrams to show the heap after each element is added.

23.6.3 Show the steps of creating a heap using {`45`, `11`, `50`, `59`, `60`, `2`, `4`, `7`, `10`}.

23.6.4 Show the heap after the root in the heap in Figure 23.15c is removed.

23.6.5 Given the following heap, show the steps of removing all nodes from the heap.



23.6.6 Which of the following statements are wrong?

```
Heap<Object> heap1 = new Heap<>();
Heap<Number> heap2 = new Heap<>();
Heap<BigInteger> heap3 = new Heap<>();
Heap<Calendar> heap4 = new Heap<>();
Heap<String> heap5 = new Heap<>();
```

23.6.7 What is the return value from invoking the `remove` method if the heap is empty?

23.6.8 What is the time complexity of inserting a new element into a heap, and what is the time complexity of deleting an element from a heap?

23.6.9 What is the height of a nonempty heap? What is the height of a heap with 16, 17, and 512 elements? If the height of a heap is 5, what is the maximum number of nodes in the heap?

23.7 Bucket and Radix Sorts

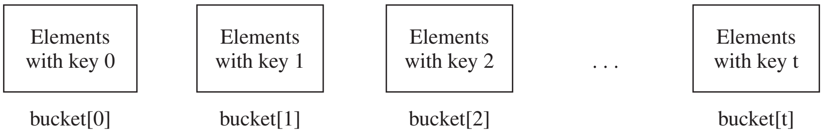
Bucket and radix sorts are efficient for sorting integers.

Key Point

All sort algorithms discussed so far are general sorting algorithms that work for any types of keys (e.g., integers, strings, and any comparable objects). These algorithms sort the elements by comparing their keys. It has been proven that no sorting algorithms based on comparisons can perform better than $O(n \log n)$. However, if the keys are integers, you can use a bucket sort without having to compare the keys.

bucket sort

The *bucket sort* algorithm works as follows. Assume the keys are in the range from `0` to `t`. We need `t + 1` buckets labeled `0`, `1`, . . . , and `t`. If an element's key is `i`, the element is put into the bucket `i`. Each bucket holds the elements with the same key value.



You can use an `ArrayList` to implement a bucket. The bucket-sort algorithm for sorting a list of elements can be described as follows:

```
public static void bucketSort(E[] list) {
  E[] bucket = (E[])new java.util.ArrayList[t+1];

  // Distribute the elements from list to buckets
  for (int i = 0; i < list.length; i++) {
    int key = list[i].getKey(); // Assume element has the getKey() method

    if (bucket[key] == null)
      bucket[key] = new java.util.ArrayList<>();

    bucket[key].add(list[i]);
  }

  // Now move the elements from the buckets back to list
  int k = 0; // k is an index for list
  for (int i = 0; i < bucket.length; i++) {
    if (bucket[i] != null) {
      for (int j = 0; j < bucket[i].size(); j++)
        list[k++] = bucket[i].get(j);
    }
  }
}
```

Clearly, it takes $O(n + t)$ time to sort the list and uses $O(n + t)$ space, where n is the list size.

stable

Note if t is too large, using the bucket sort is not desirable. Instead, you can use a radix sort. The radix sort is based on the bucket sort, but a radix sort uses only 10 buckets.

It is worthwhile to note a bucket sort is *stable*, meaning that if two elements in the original list have the same key value, their order is not changed in the sorted list. That is, if element e_1 and element e_2 have the same key and e_1 precedes e_2 in the original list, e_1 still precedes e_2 in the sorted list.

radix sort

Assume the keys are positive integers. The idea for the *radix sort* is to divide the keys into subgroups based on their radix positions. It applies a bucket sort repeatedly for the key values on radix positions, starting from the least-significant position.

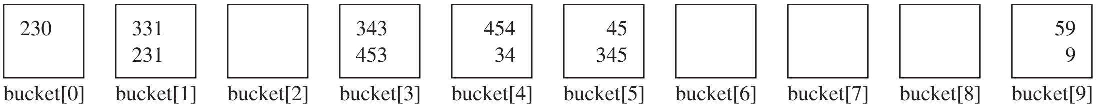
Consider sorting the elements with the following keys:

radix sort on Companion Website

```
331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9
```

queue

Apply the bucket sort on the last radix position, and the elements are put into the buckets as follows:

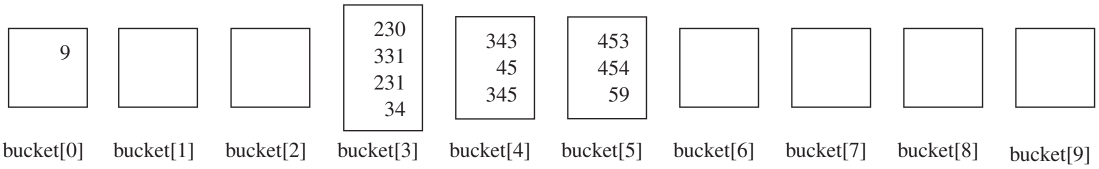


After collecting the elements from the buckets, the elements are in the following order:

`230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 9`

Apply the bucket sort on the second-to-last radix position, and the elements are put into the buckets as follows:

queue


9
230
331
231
34
343
45
345
453
454
59
bucket[0]
bucket[1]
bucket[2]
bucket[3]
bucket[4]
bucket[5]
bucket[6]
bucket[7]
bucket[8]
bucket[9]

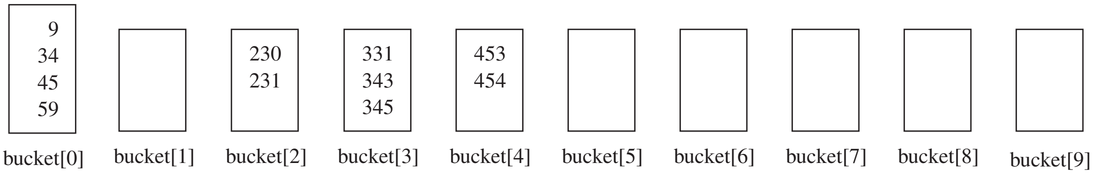
After collecting the elements from the buckets, the elements are in the following order:

`9, 230, 331, 231, 34, 343, 45, 345, 453, 454, 59`

(Note `9` is `009`.)

Apply the bucket sort on the third-to-last radix position, and the elements are put into the buckets as follows:

queue


9
34
45
59
230
231
331
343
345
453
454
bucket[0]
bucket[1]
bucket[2]
bucket[3]
bucket[4]
bucket[5]
bucket[6]
bucket[7]
bucket[8]
bucket[9]

After collecting the elements from the buckets, the elements are in the following order:

`9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454`

The elements are now sorted.

Radix sort takes *O*(*dn*) time to sort *n* elements with integer keys, where *d* is the maximum number of the radix positions among all keys.

Check Point

23.7.1 Can you sort a list of strings using a bucket sort?

23.7.2 Show how the radix sort works using the numbers **454**, **34**, **23**, **43**, **74**, **86**, and **76**.

23.8 External Sort

You can sort a large amount of data using an external sort.

Key Point

All the sort algorithms discussed in the preceding sections assume all the data to be sorted are available at one time in internal memory, such as in an array. To sort data stored in an external file, you must first bring the data to the memory then sort it internally. However, if the file is too large, all the data in the file cannot be brought to memory at one time. This section discusses how to sort data in a large external file. This is called an *external sort*.

external sort

For simplicity, assume two million `int` values are stored in a binary file named **largedata.dat**. This file was created using the program in Listing 23.11.

Listing 23.11 CreateLargeFile.java

```
import java.io.*;

public class CreateLargeFile {
  public static void main(String[] args)  throws Exception {
```

a binary output stream

output an `int` value

close output file

read an `int` value

close input file

```
    DataOutputStream output = new DataOutputStream(
      new BufferedOutputStream(
      new FileOutputStream("largedata.dat")));

    for (int i = 0; i < 2_000_000; i++)
      output.writeInt((int)(Math.random() * 1000000));

    output.close();

    // Display first 100 numbers
    DataInputStream input = new DataInputStream(
      new BufferedInputStream(new FileInputStream("largedata.dat")));
    for (int i = 0; i < 100; i++)
      System.out.print(input.readInt() + " ");

    input.close();
  }
}
```

```
569193 131317 608695 776266 767910 624915 458599 5010 ... (omitted)
```

A variation of merge sort can be used to sort this file in two phases:

Phase I: Repeatedly bring data from the file to an array, sort the array using an internal sorting algorithm, and output the data from the array to a temporary file. This process is shown in Figure 23.17. Ideally, you want to create a large array, but its maximum size depends on how much memory is allocated to the JVM by the operating system. Assume the maximum array size is 100,000 `int` values. In the temporary file, every 100,000 `int` values are sorted. They are denoted as S_1, S_2, . . . , and S_k, where the last segment, S_k, may contain less than 100,000 values.

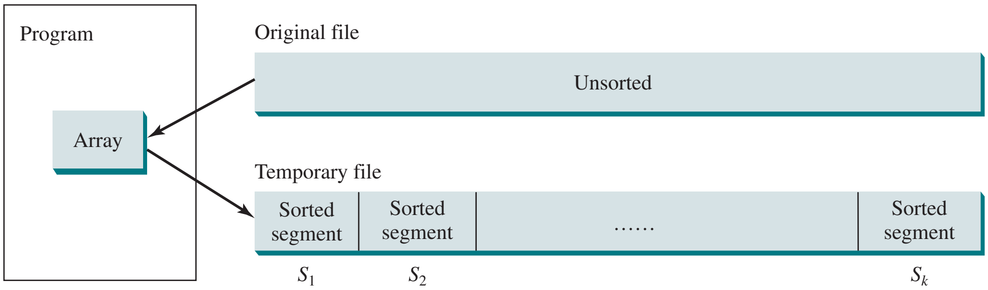


Figure 23.17 The original file is sorted in segments.

Phase II: Merge a pair of sorted segments (e.g., S_1 with S_2, S_3 with S_4, . . . , and so on) into a larger sorted segment and save the new segment into a new temporary file. Continue the same process until only one sorted segment results. Figure 23.18 shows how to merge eight segments.

Note

It is not necessary to merge two successive segments. For example, you can merge S_1 with S_5, S_2 with S_6, S_3 with S_7, and S_4 with S_8, in the first merge step. This observation is useful in implementing Phase II efficiently.

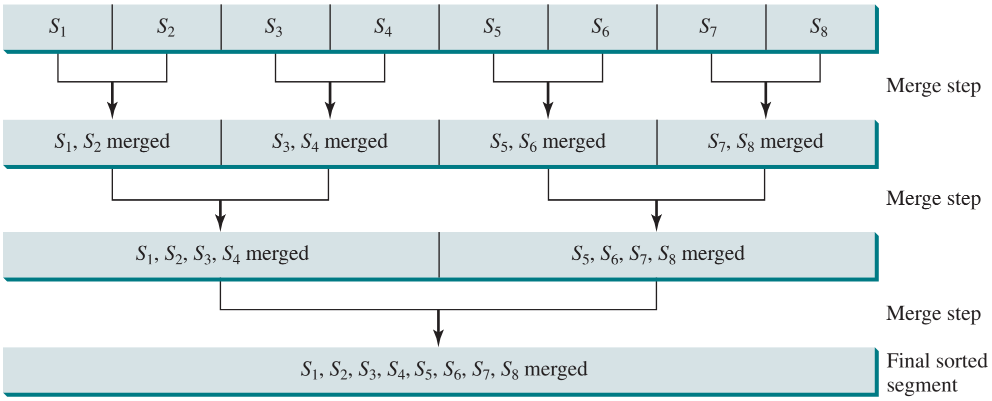

S_1
S_2
S_3
S_4
S_5
S_6
S_7
S_8
Merge step
S_1, S_2 merged
S_3, S_4 merged
S_5, S_6 merged
S_7, S_8 merged
Merge step
S_1, S_2, S_3, S_4 merged
S_5, S_6, S_7, S_8 merged
Merge step
S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8 merged
Final sorted segment

FIGURE 23.18 Sorted segments are merged iteratively.

23.8.1 Implementing Phase I

Listing 23.12 gives the method that reads each segment of data from a file, sorts the segment, and stores the sorted segments into a new file. The method returns the number of segments.

LISTING 23.12 Creating Initial Sorted Segments

```
 1  /** Sort original file into sorted segments */
 2  private static int initializeSegments
 3      (int segmentSize, String originalFile, String f1)
 4      throws Exception {
 5    int[] list = new int[segmentSize];
 6    DataInputStream input = new DataInputStream(
 7      new BufferedInputStream(new FileInputStream(originalFile)));
 8    DataOutputStream output = new DataOutputStream(
 9      new BufferedOutputStream(new FileOutputStream(f1)));
10
11    int numberOfSegments = 0;
12    while (input.available() > 0) {
13      numberOfSegments++;
14      int i = 0;
15      for ( ; input.available() > 0 && i < segmentSize; i++) {
16        list[i] = input.readInt();
17      }
18
19      // Sort an array list[0..i-1]
20      java.util.Arrays.sort(list, 0, i);
21
22      // Write the array to f1.dat
23      for (int j = 0; j < i; j++) {
24        output.writeInt(list[j]);
25      }
26    }
27
28    input.close();
29    output.close();
30
31    return numberOfSegments;
32  }
```

original file

file with sorted segments

sort a segment

output to file

close file

return # of segments

The method creates an array with the maximum size in line 5, a data input stream for the original file in line 6, and a data output stream for a temporary file in line 8. Buffered streams are used to improve performance.

Lines 14–17 read a segment of data from the file into the array. Line 20 sorts the array. Lines 23–25 write the data in the array to the temporary file.

The number of segments is returned in line 31. Note every segment has `MAX_ARRAY_SIZE` number of elements except the last segment, which may have fewer elements.

23.8.2 Implementing Phase II

In each merge step, two sorted segments are merged to form a new segment. The size of the new segment is doubled. The number of segments is reduced by half after each merge step. A segment is too large to be brought to an array in memory. To implement a merge step, copy half the number of segments from the file **f1.dat** to a temporary file **f2.dat**. Then, merge the first remaining segment in **f1.dat** with the first segment in **f2.dat** into a temporary file named **f3.dat**, as shown in Figure 23.19.

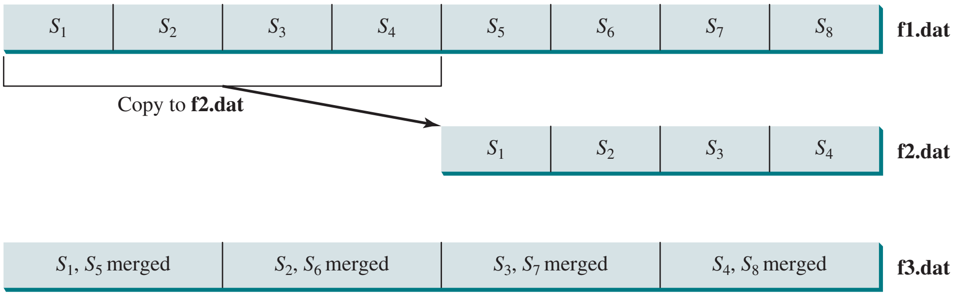


FIGURE 23.19 Sorted segments are merged iteratively.

Note

f1.dat may have one segment more than **f2.dat**. If so, move the last segment into **f3.dat** after the merge.

Listing 23.13 gives a method that copies the first half of the segments in **f1.dat** to **f2.dat**. Listing 23.14 gives a method that merges a pair of segments in **f1.dat** and **f2.dat**. Listing 23.15 gives a method that merges two segments.

LISTING 23.13 Copying First Half Segments

input stream f1
output stream f2
segments copied

```java
private static void copyHalfToF2(int numberOfSegments,
    int segmentSize, DataInputStream f1, DataOutputStream f2)
    throws Exception {
  for (int i = 0; i < (numberOfSegments / 2) * segmentSize; i++) {
    f2.writeInt(f1.readInt());
  }
}
```

LISTING 23.14 Merging All Segments

input stream f1 and f2
output stream f3
merge two segments
extra segment in f1?

```java
private static void mergeSegments(int numberOfSegments,
    int segmentSize, DataInputStream f1, DataInputStream f2,
    DataOutputStream f3)  throws Exception {
  for (int i = 0; i < numberOfSegments; i++) {
    mergeTwoSegments(segmentSize, f1, f2, f3);
  }

  // If f1 has one extra segment, copy it to f3
  while (f1.available() > 0) {
    f3.writeInt(f1.readInt());
  }
}
```

Listing 23.15 Merging Two Segments

```
private static void mergeTwoSegments(int segmentSize,
    DataInputStream f1, DataInputStream f2,
    DataOutputStream f3) throws Exception {
  int intFromF1 = f1.readInt();
  int intFromF2 = f2.readInt();
  int f1Count = 1;
  int f2Count = 1;

  while (true) {
    if (intFromF1 < intFromF2) {
      f3.writeInt(intFromF1);
      if (f1.available() == 0 || f1Count++ >= segmentSize) {
        f3.writeInt(intFromF2);
        break;
      }
      else {
        intFromF1 = f1.readInt();
      }
    }
    else {
      f3.writeInt(intFromF2);
      if (f2.available() == 0 || f2Count++ >= segmentSize) {
        f3.writeInt(intFromF1);
        break;
      }
      else {
        intFromF2 = f2.readInt();
      }
    }
  }

  while (f1.available() > 0 && f1Count++ < segmentSize) {
    f3.writeInt(f1.readInt());
  }

  while (f2.available() > 0 && f2Count++ < segmentSize) {
    f3.writeInt(f2.readInt());
  }
}
```

input stream f1 and f2

output stream f3

read from f1

read from f2

write to f3

segment in f1 finished

write to f3

segment in f2 finished

remaining f1 segment

remaining f2 segment

23.8.3 Combining Two Phases

Listing 23.16 gives the complete program for sorting `int` values in **largedata.dat** and storing the sorted data in **sortedfile.dat**.

Listing 23.16 SortLargeFile.java

```
import java.io.*;

public class SortLargeFile {
  public static final int MAX_ARRAY_SIZE = 100000;
  public static final int BUFFER_SIZE = 100000;

  public static void main(String[] args)  throws Exception {
    // Sort largedata.dat to sortedfile.dat
    sort("largedata.dat", "sortedfile.dat");

    // Display the first 100 numbers in the sorted file
```

max array size

I/O stream buffer size

create initial segments
merge recursively
merge one step
merge recursively
final sorted file
input stream `f1Input`
output stream `f2Output`
copy half segments to `f2`
close `f2Output`
input stream `f2Input`
output stream `f3Output`
merge two segments
close streams

```
    displayFile("sortedfile.dat");
  }

  /** Sort data in source file and into target file */
  public static void sort(String sourcefile, String targetfile)
      throws Exception {
    // Implement Phase 1: Create initial segments
    int numberOfSegments =
      initializeSegments(MAX_ARRAY_SIZE, sourcefile, "f1.dat");

    // Implement Phase 2: Merge segments recursively
    merge(numberOfSegments, MAX_ARRAY_SIZE,
      "f1.dat", "f2.dat", "f3.dat", targetfile);
  }

  /** Sort original file into sorted segments */
  private static int initializeSegments
      (int segmentSize, String originalFile, String f1)
      throws Exception {
    // Same as Listing 23.12, so omitted
  }

  private static void merge(int numberOfSegments, int segmentSize,
      String f1, String f2, String f3, String targetfile)
      throws Exception {
    if (numberOfSegments > 1) {
      mergeOneStep(numberOfSegments, segmentSize, f1, f2, f3);
      merge((numberOfSegments + 1) / 2, segmentSize * 2,
        f3, f1, f2, targetfile);
    }
    else {  // Rename f1 as the final sorted file
      File sortedFile = new File(targetfile);
      if (sortedFile.exists()) sortedFile.delete();
      new File(f1).renameTo(sortedFile);
    }
  }

  private static void mergeOneStep(int numberOfSegments,
      int segmentSize, String f1, String f2, String f3)
      throws Exception {
    DataInputStream f1Input = new DataInputStream(
      new BufferedInputStream(new FileInputStream(f1), BUFFER_SIZE));
    DataOutputStream f2Output = new DataOutputStream(
      new BufferedOutputStream(new FileOutputStream(f2), BUFFER_SIZE));

    // Copy half number of segments from f1.dat to f2.dat
    copyHalfToF2(numberOfSegments, segmentSize, f1Input, f2Output);
    f2Output.close();

    // Merge remaining segments in f1 with segments in f2 into f3
    DataInputStream f2Input = new DataInputStream(
      new BufferedInputStream(new FileInputStream(f2), BUFFER_SIZE));
    DataOutputStream f3Output = new DataOutputStream(
      new BufferedOutputStream(new FileOutputStream(f3), BUFFER_SIZE));

    mergeSegments(numberOfSegments / 2,
      segmentSize, f1Input, f2Input, f3Output);

    f1Input.close();
    f2Input.close();
```

```
 72       f3Output.close();
 73     }
 74 
 75     /** Copy first half number of segments from f1.dat to f2.dat */
 76     private static void copyHalfToF2(int numberOfSegments,
 77         int segmentSize, DataInputStream f1, DataOutputStream f2)
 78         throws Exception {
 79       // Same as Listing 23.13, so omitted
 80     }
 81 
 82     /** Merge all segments */
 83     private static void mergeSegments(int numberOfSegments,
 84         int segmentSize, DataInputStream f1, DataInputStream f2,
 85         DataOutputStream f3)  throws Exception {
 86       // Same as Listing 23.14, so omitted
 87     }
 88 
 89     /** Merges two segments */
 90     private static void mergeTwoSegments(int segmentSize,
 91       DataInputStream f1, DataInputStream f2,
 92       DataOutputStream f3)  throws Exception {
 93       // Same as Listing 23.15, so omitted
 94     }
 95 
 96     /** Display the first 100 numbers in the specified file */
 97     public static void displayFile(String filename) {
 98       try {
 99         DataInputStream input =
100           new DataInputStream(new FileInputStream(filename));
101         for (int i = 0; i < 100; i++)
102           System.out.print(input.readInt() + " ");
103         input.close();
104       }
105       catch (IOException ex) {
106         ex.printStackTrace();
107       }
108     }
109   }
```

display file

```
0 1 1 1 2 2 2 3 3 4 5 6 8 8 9 9 9 10 10 11...(omitted)
```

Before you run this program, first run Listing 23.11, CreateLargeFile.java, to create the file **largedata.dat**. Invoking `sort("largedata.dat", "sortedfile.dat")` (line 9) reads data from `largedata.dat` and writes sorted data to `sortedfile.dat`. Invoking `displayFile("sortedfile.dat")` (line 12) displays the first `100` numbers in the specified file. Note the files are created using binary I/O. You cannot view them using a text editor such as Notepad.

The `sort` method first creates initial segments from the original array and stores the sorted segments in a new file, **f1.dat** (lines 19–20), then produces a sorted file in `targetfile` (lines 23–24).

The `merge` method

```
merge(int numberOfSegments, int segmentSize,
  String f1, String f2, String f3, String targetfile)
```

merges the segments in `f1` into `f3` using `f2` to assist the merge. The `merge` method is invoked recursively with many merge steps. Each merge step reduces the `numberOfSegments` by half and doubles the sorted segment size. After the completion of one merge step, the next merge step merges the new segments in `f3` to `f2` using `f1` to assist the merge. The statement to invoke the new merge method is

```
merge((numberOfSegments + 1) / 2, segmentSize * 2,
  f3, f1, f2, targetfile);
```

The `numberOfSegments` for the next merge step is `(numberOfSegments + 1) / 2`. For example, if `numberOfSegments` is `5`, `numberOfSegments` is `3` for the next merge step because every two segments are merged but one is left unmerged.

The recursive `merge` method ends when `numberOfSegments` is `1`. In this case, `f1` contains sorted data. File `f1` is renamed to `targetfile` (line 45).

23.8.4 External Sort Complexity

In the external sort, the dominating cost is that of I/O. Assume n is the number of elements to be sorted in the file. In Phase I, n number of elements are read from the original file and output to a temporary file. Therefore, the I/O for Phase I is $O(n)$.

In Phase II, before the first merge step, the number of sorted segments is $\frac{n}{c}$, where c is `MAX_ARRAY_SIZE`. Each merge step reduces the number of segments by half. Thus, after the first merge step, the number of segments is $\frac{n}{2c}$. After the second merge step, the number of segments is $\frac{n}{2^2c}$, and after the third merge step the number of segments is $\frac{n}{2^3c}$. After $\log\left(\frac{n}{c}\right)$ merge steps, the number of segments is reduced to 1. Therefore, the total number of merge steps is $\log\left(\frac{n}{c}\right)$.

In each merge step, half the number of segments are read from file `f1` then written into a temporary file `f2`. The remaining segments in `f1` are merged with the segments in `f2`. The number of I/Os in each merge step is $O(n)$. Since the total number of merge steps is $\log\left(\frac{n}{c}\right)$, the total number of I/Os is

$$O(n) \times \log\left(\frac{n}{c}\right) = O(n \log n)$$

Therefore, the complexity of the external sort is $O(n \log n)$.

Check Point

23.8.1 Describe how external sort works. What is the complexity of the external sort algorithm?

23.8.2 Ten numbers {2, 3, 4, 0, 5, 6, 7, 9, 8, 1} are stored in the external file **largedata.dat**. Trace the `SortLargeFile` program by hand with `MAX_ARRAY_SIZE` 2.

KEY TERMS

bubble sort	890
bucket sort	907
complete binary tree	900
external sort	909
heap	900
heap sort	900
height of a heap	906
merge sort	892
quick sort	896
radix sort	908

CHAPTER SUMMARY

1. The worst-case complexity for a *selection sort*, *insertion sort*, *bubble sort*, and *quick sort* is $O(n^2)$.
2. The average- and worst-case complexity for a *merge sort* is $O(n \log n)$. The average time for a quick sort is also $O(n \log n)$.
3. *Heaps* are a useful data structure for designing efficient algorithms such as sorting. You learned how to define and implement a heap class, and how to insert and delete elements to/from a heap.
4. The time complexity for a *heap sort* is $O(n \log n)$.
5. *Bucket* and *radix sorts* are specialized sorting algorithms for integer keys. These algorithms sort keys using buckets rather than by comparing keys. They are more efficient than general sorting algorithms.
6. A variation of the merge sort—called an *external sort*—can be applied to sort large amounts of data from external files.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

Sections 23.3–23.5

23.1 (*Generic bubble sort*) Write the following two generic methods using bubble sort. The first method sorts the elements using the `Comparable` interface, and the second uses the `Comparator` interface.

```
public static <E extends Comparable<E>>
  void bubbleSort(E[] list)
public static <E> void bubbleSort(E[] list,
  Comparator<? super E> comparator)
```

23.2 (*Generic merge sort*) Write the following two generic methods using merge sort. The first method sorts the elements using the `Comparable` interface and the second uses the `Comparator` interface.

```
public static <E extends Comparable<E>>
  void mergeSort(E[] list)
public static <E> void mergeSort(E[] list,
  Comparator<? super E> comparator)
```

23.3 (*Generic quick sort*) Write the following two generic methods using quick sort. The first method sorts the elements using the `Comparable` interface, and the second uses the `Comparator` interface.

```
public static <E extends Comparable<E>>
  void quickSort(E[] list)
public static <E> void quickSort(E[] list,
  Comparator<? super E> comparator)
```

23.4 (*Improve quick sort*) The quick-sort algorithm presented in the book selects the first element in the list as the pivot. Revise it by selecting the median among the first, middle, and the last elements in the list.

***23.5** (*Modify merge sort*) Rewrite the `mergeSort` method to recursively sort the first half of the array and the second half of the array without creating new temporary arrays, then merge the two into a temporary array and copy its contents to the original array, as shown in Figure 23.6b.

23.6 (*Check order*) Write the following overloaded methods that check whether an array is ordered in ascending order or descending order. By default, the method checks ascending order. To check descending order, pass `false` to the ascending argument in the method.

```
public static boolean ordered(int[] list)
public static boolean ordered(int[] list, boolean ascending)
public static boolean ordered(double[] list)
public static boolean ordered
  (double[] list, boolean ascending)
public static <E extends Comparable<E>>
  boolean ordered(E[] list)
public static <E extends Comparable<E>> boolean ordered
  (E[] list, boolean ascending)
public static <E> boolean ordered(E[] list,
  Comparator<? super E> comparator)
public static <E> boolean ordered(E[] list,
  Comparator<? super E> comparator, boolean ascending)
```

Section 23.6

max-heap
min-heap

23.7 (*Min-heap*) The heap presented in the text is also known as a *max-heap*, in which each node is greater than or equal to any of its children. A *min-heap* is a heap in which each node is less than or equal to any of its children. Min-heaps are often used to implement priority queues. Revise the `Heap` class in Listing 23.9 to implement a min-heap.

23.8 (*Generic insertion sort*) Write the following two generic methods using insertion sort. The first method sorts the elements using the `Comparable` interface, and the second uses the `Comparator` interface.

```
public static <E extends Comparable<E>>
  void insertionSort(E[] list)
public static <E> void insertionSort(E[] list,
  Comparator<? super E> comparator)
```

***23.9** (*Generic heap sort*) Write the following two generic methods using heap sort. The first method sorts the elements using the `Comparable` interface, and the second uses the `Comparator` interface. (*Hint*: Use the `Heap` class in Programming Exercise 23.5.)

```
public static <E extends Comparable<E>>
  void heapSort(E[] list)
public static <E> void heapSort(E[] list,
  Comparator<? super E> comparator)
```

****23.10** (*Heap visualization*) Write a program that displays a heap graphically, as shown in Figure 23.10. The program lets you insert and delete an element from the heap.

23.11 (*Heap `clone` and `equals`*) Implement the `clone` and `equals` method in the `Heap` class.
Use the template at `liveexample.pearsoncmg.com/test/Exercise23_11.txt` for your code.

Section 23.7

***23.12** (*Radix sort*) Write a program that randomly generates 1,000,000 integers and sorts them using radix sort.

***23.13** (*Execution time for sorting*) Write a program that obtains the execution time of selection sort, bubble sort, merge sort, quick sort, heap sort, and radix sort for input size 50,000, 100,000, 150,000, 200,000, 250,000, and 300,000. Your program should create data randomly and print a table like this:

Array size	Selection Sort	Bubble Sort	Merge Sort	Quick Sort	Heap Sort	Radix Sort
50,000						
100,000						
150,000						
200,000						
250,000						
300,000						

(*Hint*: You can use the following code template to obtain the execution time.)

```
long startTime = System.nanoTime();
perform the task;
long endTime = System.nanoTime();
long executionTime = endTime - startTime;
```

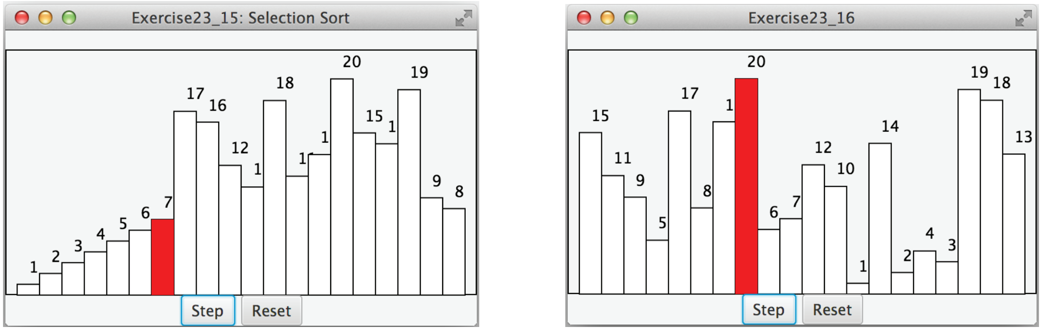
Section 23.8

***23.14** (*Execution time for external sorting*) Write a program that obtains the execution time of external sorts for integers of size 5,000,000, 10,000,000, 15,000,000, 20,000,000, 25,000,000, and 30,000,000. Your program should print a table like this:

File size	5,000,000	10,000,000	15,000,000	20,000,000	25,000,000	30,000,000
Time						

Comprehensive

***23.15** (*Selection-sort animation*) Write a program that animates the selection-sort algorithm. Create an array that consists of 20 distinct numbers from 1 to 20 in a random order. The array elements are displayed in a histogram, as shown in Figure 23.20a. Clicking the *Step* button causes the program to perform an iteration of the outer loop in the algorithm and repaints the histogram for the new array. Color the last bar in the sorted subarray. When the algorithm is finished, display a message to inform the user. Clicking the *Reset* button creates a new random array for a new start. (You can easily modify the program to animate the insertion algorithm.)



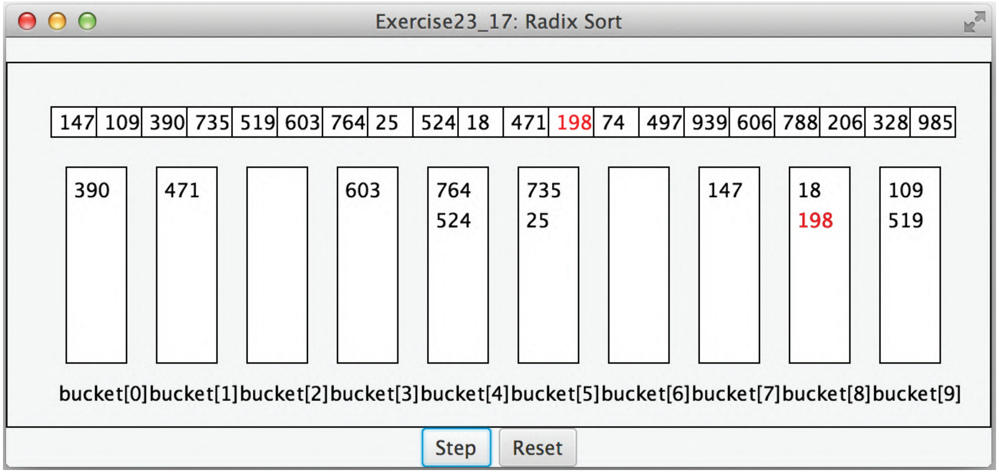
(a)

(b)

FIGURE 23.20 (a) The program animates selection sort. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) The program animates bubble sort.

***23.16** (*Bubble-sort animation*) Write a program that animates the bubble-sort algorithm. Create an array that consists of 20 distinct numbers from 1 to 20 in a random order. The array elements are displayed in a histogram, as shown in Figure 23.20b. Clicking the *Step* button causes the program to perform one comparison in the algorithm and repaints the histogram for the new array. Color the bar that represents the number being considered in the swap. When the algorithm is finished, display a message to inform the user. Clicking the *Reset* button creates a new random array for a new start.

***23.17** (*Radix-sort animation*) Write a program that animates the radix-sort algorithm. Create an array that consists of 20 random numbers from 0 to 1,000. The array elements are displayed, as shown in Figure 23.21. Clicking the *Step* button causes the program to place a number in a bucket. The number that has just been placed is displayed in red. Once all the numbers are placed in the buckets, clicking the *Step* button collects all the numbers from the buckets and moves them back to the array. When the algorithm is finished, clicking the *Step* button displays a message to inform the user. Clicking the *Reset* button creates a new random array for a new start.

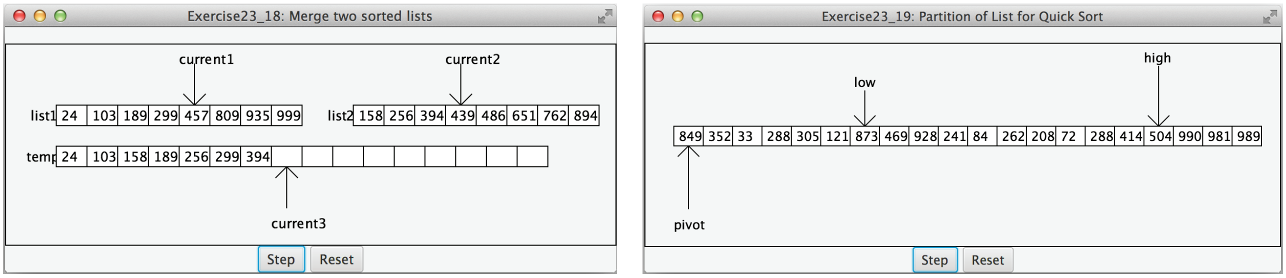


147	109	390	735	519	603	764	25	524	18	471	198	74	497	939	606	788	206	328	985

bucket[0]	bucket[1]	bucket[2]	bucket[3]	bucket[4]	bucket[5]	bucket[6]	bucket[7]	bucket[8]	bucket[9]
390	471		603	764 524	735 25		147	18 198	109 519

FIGURE 23.21 The program animates radix sort. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***23.18** (*Merge animation*) Write a program that animates the merge of two sorted lists. Create two arrays, `list1` and `list2`, each of which consists of 8 random numbers from 1 to 999. The array elements are displayed, as shown in Figure 23.22a. Clicking the *Step* button causes the program to move an element from `list1` or `list2` to `temp`. Clicking the *Reset* button creates two new random arrays for a new start. When the algorithm is finished, clicking the *Step* button displays a message to inform the user.


Exercise23_18: Merge two sorted lists
current1
current2
list1 24 103 189 299 457 809 935 999
list2 158 256 394 439 486 651 762 894
temp 24 103 158 189 256 299 394
current3
Step
Reset
Exercise23_19: Partition of List for Quick Sort
high
low
849 352 33 288 305 121 873 469 928 241 84 262 208 72 288 414 504 990 981 989
pivot
Step
Reset

(a) (b)

FIGURE 23.22 The program animates a merge of two sorted lists. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) The program animates a partition for quick sort.

***23.19** (*Quick-sort partition animation*) Write a program that animates the partition for a quick sort. The program creates a list that consists of 20 random numbers from 1 to 999. The list is displayed, as shown in Figure 23.22b. Clicking the *Step* button causes the program to move `low` to the right or `high` to the left, or swap the elements at `low` and `high`. Clicking the *Reset* button creates a new list of random numbers for a new start. When the algorithm is finished, clicking the *Step* button displays a message to inform the user.

CHAPTER

24

IMPLEMENTING LISTS, STACKS, QUEUES, AND PRIORITY QUEUES

Objectives

- To design common operations of lists in an interface and make the interface a subtype of `Collection` (§24.2).
- To design and implement an array list using an array (§24.3).
- To design and implement a linked list using a linked structure (§24.4).
- To design and implement a stack class using an array list and a queue class using a linked list (§24.5).
- To design and implement a priority queue using a heap (§24.6).

24.1 Introduction

Key Point

This chapter focuses on implementing data structures.

Lists, stacks, queues, and priority queues are classic data structures typically covered in a data structures course. They are supported in the Java API, and their uses were presented in Chapter 20, Lists, Stacks, Queues, and Priority Queues. This chapter will examine how these data structures are implemented under the hood. Implementation of sets and maps will be covered in Chapter 27. Through these implementations, you will gain valuable insight on data structures and learn how to design and implement custom data structures.

24.2 Common Operations for Lists

Key Point

Common operations of lists are defined in the `List` *interface.*

A list is a popular data structure for storing data in sequential order—for example, a list of students, a list of available rooms, a list of cities, and a list of books. You can perform the following operations on a list:

- Retrieve an element from the list.
- Insert a new element into the list.
- Delete an element from the list.
- Find out how many elements are in the list.
- Determine whether an element is in the list.
- Check whether the list is empty.

There are two ways to implement a list. One is to use an *array* to store the elements. Array size is fixed. If the capacity of the array is exceeded, you need to create a new, larger array and copy all the elements from the current array to the new array. The other approach is to use a *linked structure*. A linked structure consists of nodes. Each node is dynamically created to hold an element. All the nodes are linked together to form a list. Thus, you can define two classes for lists. For convenience, let's name these two classes `MyArrayList` and `MyLinkedList`. These two classes have common operations but different implementations.

For an interactive demo on how array lists and linked lists work, see https://liveexample.pearsoncmg.com/dsanimation/ArrayListeBook.html and https://liveexample.pearsoncmg.com/dsanimation/LinkedListeBook.html, as shown in Figure 24.1.

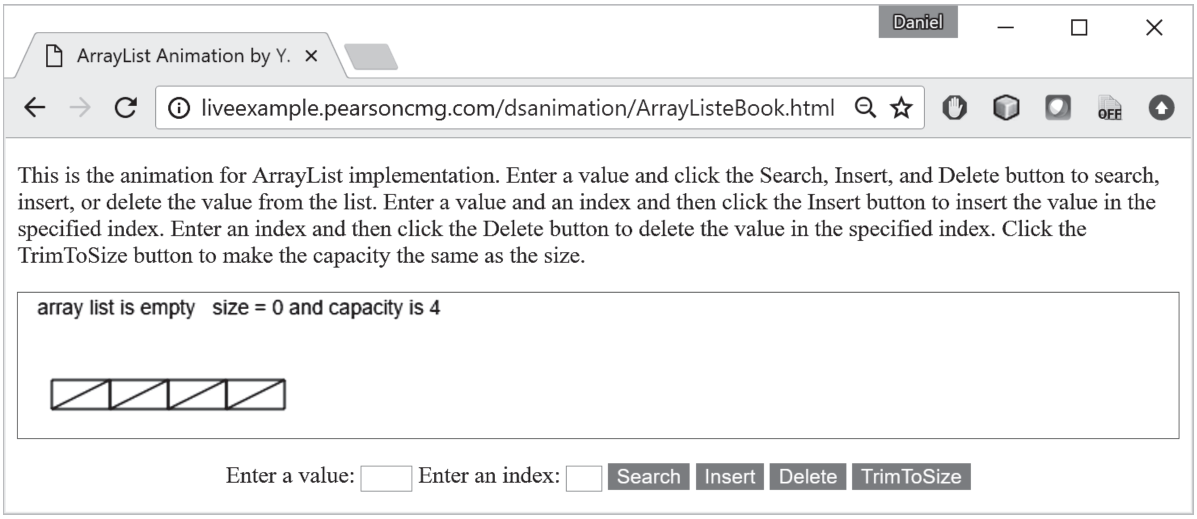
Design Guide

default methods in interfaces

Prior to Java 8, a popular design strategy for Java data structures is to define common operations in interfaces and provide convenient abstract classes for partially implementing the interfaces. So, the concrete classes can simply extend the convenient abstract classes without implementing the full interfaces. Java 8 enables you to define default methods. You can provide default implementation for some of the methods in the interfaces rather than in convenient abstract classes. Using default methods eliminate the need for convenient abstract classes.

list animation on Companion Website

Let us name the interface `MyList` and define it as a subtype of `Collection` so the common operations in the `Collection` interface are also available in `MyList`. Figure 24.2 shows the relationship of `Collection`, `MyList`, `MyArrayList`, and `MyLinkedList`. The methods in `MyList` are shown in Figure 24.3. Listing 24.1 gives the source code for `MyList`.

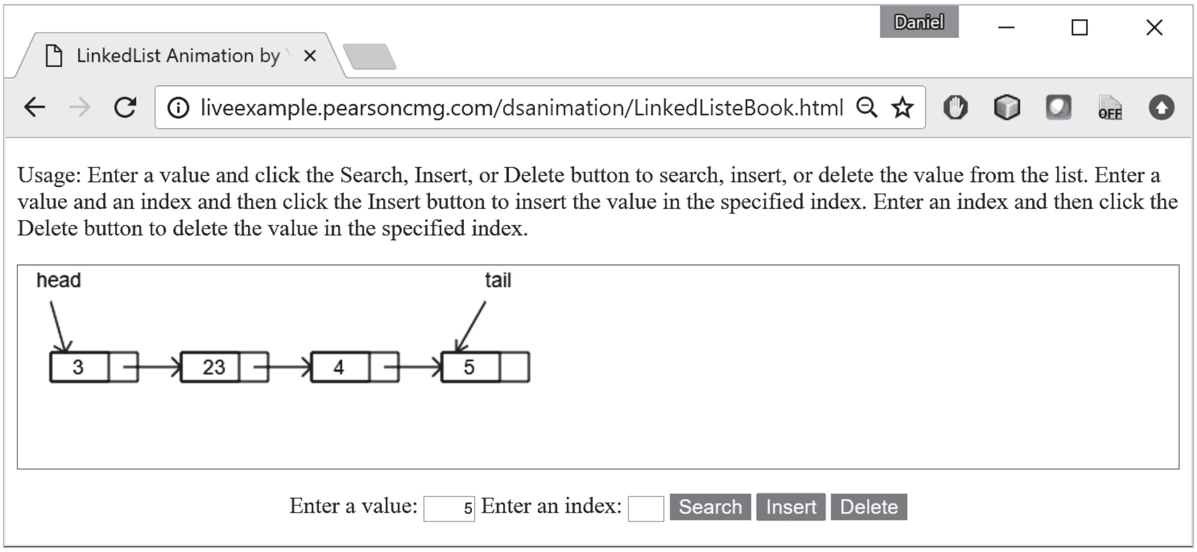

Daniel
ArrayList Animation by Y.
liveexample.pearsoncmg.com/dsanimation/ArrayListeBook.html
Search
Insert
Delete
TrimToSize

This is the animation for ArrayList implementation. Enter a value and click the Search, Insert, and Delete button to search, insert, or delete the value from the list. Enter a value and an index and then click the Insert button to insert the value in the specified index. Enter an index and then click the Delete button to delete the value in the specified index. Click the TrimToSize button to make the capacity the same as the size.

array list is empty size = 0 and capacity is 4

Enter a value: Enter an index:

(a) ArrayList animation. *Source* Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.


Daniel
LinkedList Animation by
liveexample.pearsoncmg.com/dsanimation/LinkedListeBook.html
head
tail
3
23
4
5
Search
Insert
Delete

Usage: Enter a value and click the Search, Insert, or Delete button to search, insert, or delete the value from the list. Enter a value and an index and then click the Insert button to insert the value in the specified index. Enter an index and then click the Delete button to delete the value in the specified index.

Enter a value: 5 Enter an index:

(b) LinkedList animation

FIGURE 24.1 The animation tool enables you to see how array lists and linked lists work.

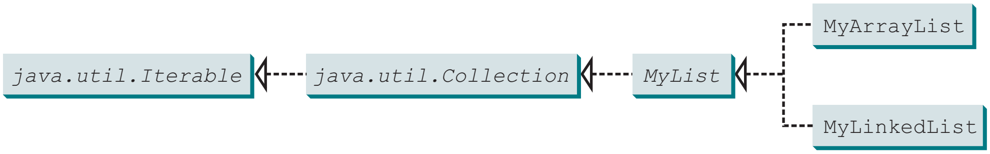

java.util.Iterable
java.util.Collection
MyList
MyArrayList
MyLinkedList

FIGURE 24.2 `MyList` defines a common interface for `MyArrayList` and `MyLinkedList`.

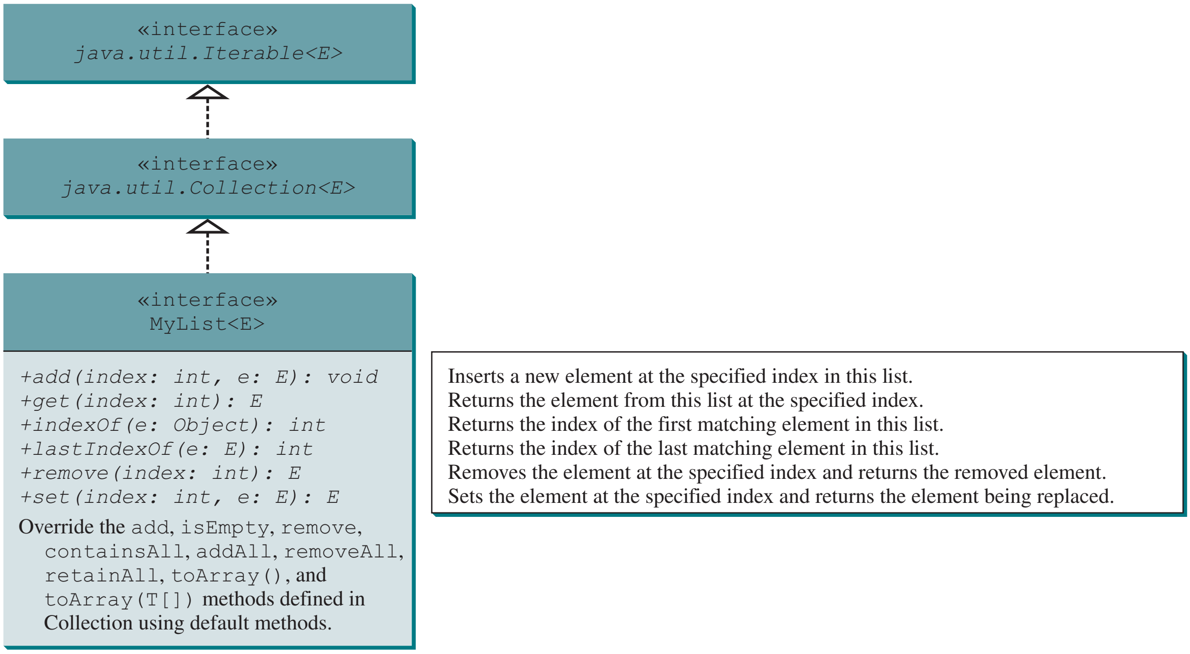

«interface»
java.util.Iterable<E>
«interface»
java.util.Collection<E>
«interface»
MyList<E>
+add(index: int, e: E): void
+get(index: int): E
+indexOf(e: Object): int
+lastIndexOf(e: E): int
+remove(index: int): E
+set(index: int, e: E): E
Override the add, isEmpty, remove, containsAll, addAll, removeAll, retainAll, toArray(), and toArray(T[]) methods defined in Collection using default methods.
Inserts a new element at the specified index in this list.
Returns the element from this list at the specified index.
Returns the index of the first matching element in this list.
Returns the index of the last matching element in this list.
Removes the element at the specified index and returns the removed element.
Sets the element at the specified index and returns the element being replaced.

FIGURE 24.3 `MyList` defines the methods for manipulating a list and partially implements some of the methods defined in the `Collection` interface.

LISTING 24.1 MyList.java

```java
import java.util.Collection;

public interface MyList<E> extends Collection<E> {
  /** Add a new element at the specified index in this list */
  public void add(int index, E e);

  /** Return the element from this list at the specified index */
  public E get(int index);

  /** Return the index of the first matching element in this list.
   *  Return -1 if no match. */
  public int indexOf(Object e);

  /** Return the index of the last matching element in this list
   *  Return -1 if no match. */
  public int lastIndexOf(E e);

  /** Remove the element at the specified position in this list
   *  Shift any subsequent elements to the left.
   *  Return the element that was removed from the list. */
  public E remove(int index);

  /** Replace the element at the specified position in this list
   *  with the specified element and returns the new set. */
  public E set(int index, E e);

  @Override /** Add a new element at the end of this list */
  public default boolean add(E e) {
    add(size(), e);
    return true;
```

add(index, e)
get(index)
indexOf(e)
lastIndexOf(e)
remove(e)
set(index, e)
default add(e)

```
  }

  @Override /** Return true if this list contains no elements */
  public default boolean isEmpty() {
    return size() == 0;
  }

  @Override /** Remove the first occurrence of the element e
   * from this list. Shift any subsequent elements to the left.
   * Return true if the element is removed. */
  public default boolean remove(Object e) {
    if (indexOf(e) >= 0) {
      remove(indexOf(e));
      return true;
    }
    else
      return false;
  }

  @Override
  public default boolean containsAll(Collection<?> c) {
    // Left as an exercise
    return true;
  }

  @Override
  public default boolean addAll(Collection<?  extends E> c) {
    // Left as an exercise
    return true;
  }

  @Override
  public default boolean removeAll(Collection<?> c) {
    // Left as an exercise
    return true;
  }

  @Override
  public default boolean retainAll(Collection<?> c) {
    // Left as an exercise
    return true;
  }

  @Override
  public default Object[] toArray() {
    // Left as an exercise
    return null;
  }

  @Override
  public default <T> T[] toArray(T[] array) {
    // Left as an exercise
    return null;
  }
}
```

default `isEmpty()`

implement `remove(E e)`

implement `containsAll`

implement `addAll`

implement `removeAll`

implement `retainAll`

implement `toArray()`

implement `toArray(T[])`

The methods **`isEmpty()`**, **`add(E)`**, **`remove(E)`**, **`containsAll`**, **`addAll`**, **`removeAll`**, **`retainAll`**, **`toArray()`**, and **`toArray(T[])`** are defined in the **`Collection`** interface. Since these methods are implementable in **`MyList`**, they are overridden in the **`MyList`** interface as default methods. The implementation for **`isEmpty()`**, **`add(E)`**, and **`remove(E)`** are provided and the implementation for other default methods are left as exercises in Programming Exercise 24.1.

The following sections give the implementation for `MyArrayList` and `MyLinkedList`, respectively.

Check Point

24.2.1 Suppose `list` is an instance of `MyList`, can you get an iterator for list using `list.iterator()`?

24.2.2 Can you create a list using `new MyList()`?

24.2.3 What methods in `Collection` are overridden as default methods in `MyList`?

24.2.4 What are the benefits of overriding the methods in `Collection` as default methods in `MyList`?

24.3 Array Lists

Key Point

An array list is implemented using an array.

An array is a fixed-size data structure. Once an array is created, its size cannot be changed. Nevertheless, you can still use arrays to implement dynamic data structures. The trick is to create a larger new array to replace the current array, if the current array cannot hold new elements in the list.

Initially, an array, say `data` of `E[]` type, is created with a default size. When inserting a new element into the array, first make sure there is enough room in the array. If not, create a new array twice as large as the current one. Copy the elements from the current array to the new array. The new array now becomes the current array. Before inserting a new element at a specified index, shift all the elements after the index to the right and increase the list size by `1`, as shown in Figure 24.4.

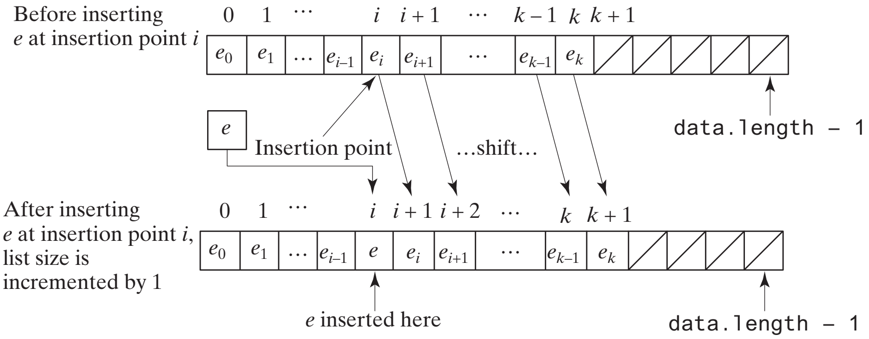

Before inserting
e at insertion point i
0 1 ... i i + 1 ... k – 1 k k + 1
e_0 e_1 ... e_{i-1} e_i e_{i+1} ... e_{k-1} e_k
e
Insertion point
...shift...
data.length – 1
After inserting
e at insertion point i,
list size is
incremented by 1
0 1 ... i i + 1 i + 2 ... k k + 1
e_0 e_1 ... e_{i-1} e e_i e_{i+1} ... e_{k-1} e_k
e inserted here
data.length – 1

FIGURE 24.4 Inserting a new element into the array requires that all the elements after the insertion point be shifted one position to the right, so the new element can be inserted at the insertion point.

Note

The data array is of type `E[]`. Each cell in the array actually stores the reference of an object.

To remove an element at a specified index, shift all the elements after the index to the left by one position and decrease the list size by `1`, as shown in Figure 24.5.

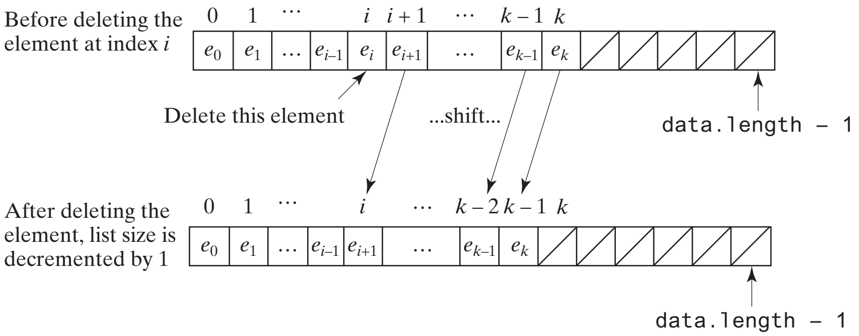

Before deleting the element at index i
0 1 … i $i+1$ … $k-1$ k
e_0 e_1 … e_{i-1} e_i e_{i+1} … e_{k-1} e_k
Delete this element
...shift...
data.length – 1
After deleting the element, list size is decremented by 1
0 1 … i … $k-2$ $k-1$ k
e_0 e_1 … e_{i-1} e_{i+1} … e_{k-1} e_k
data.length – 1

FIGURE 24.5 Deleting an element from the array requires that all the elements after the deletion point be shifted one position to the left.

`MyArrayList` uses an array to implement `MyList`, as shown in Figure 24.6. Its implementation is given in Listing 24.2.

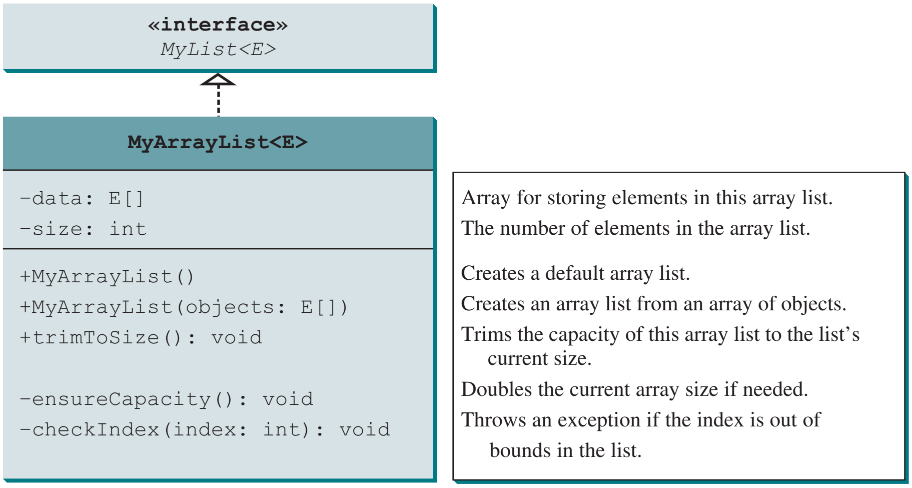

«interface»
MyList<E>
MyArrayList<E>
-data: E[]
-size: int
+MyArrayList()
+MyArrayList(objects: E[])
+trimToSize(): void
-ensureCapacity(): void
-checkIndex(index: int): void
Array for storing elements in this array list.
The number of elements in the array list.
Creates a default array list.
Creates an array list from an array of objects.
Trims the capacity of this array list to the list's current size.
Doubles the current array size if needed.
Throws an exception if the index is out of bounds in the list.

FIGURE 24.6 `MyArrayList` implements a list using an array.

LISTING 24.2 MyArrayList.java

```
public class MyArrayList<E> implements MyList<E> {
  public static final int INITIAL_CAPACITY = 16;
  private E[] data = (E[])new Object[INITIAL_CAPACITY];
  private int size = 0; // Number of elements in the list

  /** Create an empty list */
  public MyArrayList() {
  }

  /** Create a list from an array of objects */
```

initial capacity
create an array
number of elements
no-arg constructor

constructor

```
  public MyArrayList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      add(objects[i]); // Warning: don't use super(objects)!
  }

  @Override /** Add a new element at the specified index */
```

add

```
  public void add(int index, E e) {
    // Ensure the index is in the right range
    if (index < 0 || index > size)
      throw new IndexOutOfBoundsException
        ("Index: " + index + ", Size: " + size);

    ensureCapacity();

    // Move the elements to the right after the specified index
    for (int i = size - 1; i >= index; i--)
      data[i + 1] = data[i];

    // Insert new element to data[index]
    data[index] = e;

    // Increase size by 1
    size++;
  }

  /** Create a new larger array, double the current size + 1 */
```

ensureCapacity

```
  private void ensureCapacity() {
    if (size >= data.length) {
```

double capacity + 1

```
      E[] newData = (E[])(new Object[size * 2 + 1]);
      System.arraycopy(data, 0, newData, 0, size);
      data = newData;
    }
  }

  @Override /** Clear the list */
```

clear

```
  public void clear() {
    data = (E[])new Object[INITIAL_CAPACITY];
    size = 0;
  }

  @Override /** Return true if this list contains the element */
```

contains

```
  public boolean contains(Object e) {
    for (int i = 0; i < size; i++)
      if (e.equals(data[i])) return true;

    return false;
  }

  @Override /** Return the element at the specified index */
```

get

```
  public E get(int index) {
    checkIndex(index);
    return data[index];
  }

```

checkIndex

```
  private void checkIndex(int index) {
    if (index < 0 || index >= size)
      throw new IndexOutOfBoundsException
        ("Index: " + index + ", Size: " + size);
  }

```

indexOf

```
  @Override /** Return the index of the first matching element
   * in this list. Return -1 if no match. */
  public int indexOf(Object e) {
    for (int i = 0; i < size; i++)
      if (e.equals(data[i]))  return i;

    return -1;
  }
```

lastIndexOf

```
  @Override /** Return the index of the last matching element
   * in this list. Return -1 if no match. */
  public int lastIndexOf(E e) {
    for (int i = size - 1; i >= 0; i--)
      if (e.equals(data[i])) return i;

    return -1;
  }
```

remove

```
  @Override /** Remove the element at the specified position
   * in this list. Shift any subsequent elements to the left.
   * Return the element that was removed from the list. */
  public E remove(int index) {
    checkIndex(index);

    E e = data[index];

    // Shift data to the left
    for (int j = index; j < size - 1; j++)
      data[j] = data[j + 1];

    data[size - 1] = null; // This element is now null

    // Decrement size
    size--;

    return e;
  }
```

set

```
  @Override /** Replace the element at the specified position
   * in this list with the specified element. */
  public E set(int index, E e) {
    checkIndex(index);
    E old = data[index];
    data[index] = e;
    return old;
  }
```

toString

```
  @Override
  public String toString() {
    StringBuilder result = new StringBuilder("[");

    for (int i = 0; i < size; i++) {
      result.append(data[i]);
      if (i < size - 1) result.append(", ");
    }

    return result.toString() + "]";
  }

  /** Trims the capacity to current size */
```

```
trimToSize        131    public void trimToSize() {
                  132      if (size != data.length) {
                  133        E[] newData = (E[])(new Object[size]);
                  134        System.arraycopy(data, 0, newData, 0, size);
                  135        data = newData;
                  136      }  // If size == capacity, no need to trim
                  137    }
                  138
                  139    @Override /** Override iterator() defined in Iterable */
iterator          140    public java.util.Iterator<E> iterator() {
                  141      return new ArrayListIterator();
                  142    }
                  143
                  144    private class ArrayListIterator
                  145        implements java.util.Iterator<E> {
                  146      private int current = 0; // Current index
                  147
                  148      @Override
                  149      public boolean hasNext() {
                  150        return current < size;
                  151      }
                  152
                  153      @Override
                  154      public E next() {
                  155        return data[current++];
                  156      }
                  157
                  158      @Override // Remove the element returned by the last next()
                  159      public void remove() {
                  160        if (current == 0)  // next() has not been called yet
                  161          throw new IllegalStateException();
                  162        MyArrayList.this.remove(--current);
                  163      }
                  164    }
                  165
                  166    @Override /** Return the number of elements in this list */
size              167    public int size() {
                  168      return size;
                  169    }
                  170  }
```

The constant `INITIAL_CAPACITY` (line 2) is used to create an initial array `data` (line 3). Owing to generics type erasure (see Restriction 2 in Section 19.8), you cannot create a generic array using the syntax `new e[INITIAL_CAPACITY]`. To circumvent this limitation, an array of the `Object` type is created in line 3 and cast into `E[]`. The `size` data field tracks the number of elements in the list (line 4).

add

The `add(int index, E e)` method (lines 17–34) inserts the element `e` at the specified `index` in the array. This method first invokes `ensureCapacity()` (line 23), which ensures that there is a space in the array for the new element. It then shifts all the elements after the index one position to the right before inserting the element (lines 26 and 27). After the element is added, `size` is incremented by `1` (line 33).

ensureCapacity

The `ensureCapacity()` method (lines 37–43) checks whether the array is full. If so, the program creates a new array that doubles the current array size + `1`, copies the current array to the new array using the `System.arraycopy` method, and sets the new array as the current array. Note the current size might be `0` after invoking the `trimToSize()` method. `new Object[2 * size + 1]` (line 39) ensures that the new size is not `0`.

clear

The `clear()` method (lines 46–49) creates a new array using the size as `INITIAL_CAPACITY` and resets the variable `size` to `0`. The class will work if line 47 is deleted. However, the class will have a memory leak because the elements are still in the array, although they are no longer needed. By creating a new array and assigning it to `data`, the old array and the elements stored in the old array become garbage, which will be automatically collected by the JVM.

The `contains(Object e)` method (lines 52–57) checks whether element `e` is contained in the array by comparing `e` with each element in the array using the `equals` method.

`contains`

The `get(int index)` method (lines 60–63) checks if `index` is within the range and returns `data[index]` if `index` is in the range.

The `checkIndex(int index)` method (lines 65–69) checks if `index` is within the range. If not, the method throws an `IndexOutOfBoundsException` (line 67).

`checkIndex`

The `indexOf(Object e)` method (lines 73–78) compares element `e` with the elements in the array, starting from the first one. If a match is found, the index of the element is returned; otherwise, `-1` is returned.

`indexOf`

The `lastIndexOf(Object e)` method (lines 82–87) compares element `e` with the elements in the array, starting from the last one. If a match is found, the index of the element is returned; otherwise, `-1` is returned.

`lastIndexOf`

The `remove(int index)` method (lines 92–107) shifts all the elements after the index one position to the left (lines 98 and 99) and decrements `size` by `1` (line 104). The last element is not used anymore and is set to `null` (line 101).

`remove`

The `set(int index, E e)` method (lines 111–116) simply assigns `e` to `data[index]` to replace the element at the specified index with element `e`.

`set`

The `toString()` method (lines 119–128) overrides the `toString` method in the `Object` class to return a string representing all the elements in the list.

`toString`

The `trimToSize()` method (lines 131–137) creates a new array whose size matches the current array-list size (line 133), copies the current array to the new array using the `System.arraycopy` method (line 134), and sets the new array as the current array (line 135). Note if `size == capacity`, there is no need to trim the size of the array.

`trimToSize`

The `iterator()` method defined in the `java.lang.Iterable` interface is implemented to return an instance on `java.util.Iterator` (lines 140–142). The `ArrayListIterator` class implements `Iterator` with concrete methods for `hasNext`, `next`, and `remove` (lines 144–164). It uses `current` to denote the current position of the element being traversed (line 146).

`iterator`

The `size()` method simply returns the number of elements in the array list (lines 167–169).

`size()`

Listing 24.3 gives an example that creates a list using `MyArrayList`. It uses the `add` method to add strings to the list, and the `remove` method to remove strings. Since `MyArrayList` implements `Iterable`, the elements can be traversed using a foreach loop (lines 35 and 36).

LISTING 24.3 `TestMyArrayList.java`

```
 1  public class TestMyArrayList {
 2    public static void main(String[] args) {
 3      // Create a list
 4      MyList<String> list = new MyArrayList<>();
 5
 6      // Add elements to the list
 7      list.add("America"); // Add it to the list
 8      System.out.println("(1) " + list);
 9
10      list.add(0, "Canada"); // Add it to the beginning of the list
11      System.out.println("(2) " + list);
12
13      list.add("Russia"); // Add it to the end of the list
14      System.out.println("(3) " + list);
15
16      list.add("France"); // Add it to the end of the list
17      System.out.println("(4) " + list);
```

create a list

add to list

remove from list

use iterator

```

    list.add(2, "Germany"); // Add it to the list at index 2
    System.out.println("(5) " + list);

    list.add(5, "Norway"); // Add it to the list at index 5
    System.out.println("(6) " + list);

    // Remove elements from the list
    list.remove("Canada"); // Same as list.remove(0) in this case
    System.out.println("(7) " + list);

    list.remove(2); // Remove the element at index 2
    System.out.println("(8) " + list);

    list.remove(list.size() - 1); // Remove the last element
    System.out.print("(9) " + list + "\n(10) ");

    for (String s: list)
      System.out.print(s.toUpperCase() + " ");
  }
}
```

```
(1) [America]
(2) [Canada, America]
(3) [Canada, America, Russia]
(4) [Canada, America, Russia, France]
(5) [Canada, America, Germany, Russia, France]
(6) [Canada, America, Germany, Russia, France, Norway]
(7) [America, Germany, Russia, France, Norway]
(8) [America, Germany, France, Norway]
(9) [America, Germany, France]
(10) AMERICA GERMANY FRANCE
```

Check Point

24.3.1 What are the limitations of the array data type?

24.3.2 `MyArrayList` is implemented using an array, and an array is a fixed-size data structure. Why is `MyArrayList` considered a dynamic data structure?

24.3.3 Show the length of the array in `MyArrayList` after each of the following statements is executed:

```
MyArrayList<Double> list = new MyArrayList<>();
list.add(1.5);
list.trimToSize();
list.add(3.4);
list.add(7.4);
list.add(17.4);
```

24.3.4 What is wrong if lines 11 and 12 in Listing 24.2, MyArrayList.java,

```
for (int i = 0; i < objects.length; i++)
  add(objects[i]);
```

are replaced by

```
data = objects;
size = objects.length;
```

24.3.5 If you change the code in line 33 in Listing 24.2, MyArrayList.java, from

```
E[] newData = (E[])(new Object[size * 2 + 1]);
```

to

```
E[] newData = (E[])(new Object[size * 2]);
```

the program is incorrect. Can you find the reason?

24.3.6 Will the `MyArrayList` class have memory leak if the following code in line 41 is deleted?

```
data = (E[])new Object[INITIAL_CAPACITY];
```

24.3.7 The **`get(index)`** method invokes the **`checkIndex(index)`** method (lines 59–63 in Listing 24.2) to throw an **`IndexOutOfBoundsException`** if the index is out of bounds. Suppose the **`add(index, e)`** method is implemented as follows:

```
public void add(int index, E e) {
  checkIndex(index);

  // Same as lines 23-33 in Listing 24.2 MyArrayList.java
}
```

What will happen if you run the following code?

```
MyArrayList<String> list = new MyArrayList<>();
list.add("New York");
```

24.4 Linked Lists

A linked list is implemented using a linked structure.

Key Point

Since **`MyArrayList`** is implemented using an array, the methods **`get(int index)`** and **`set(int index, E e)`** for accessing and modifying an element through an index and the **`add(E e)`** method for adding an element at the end of the list are efficient. However, the methods **`add(int index, E e)`** and **`remove(int index)`** are inefficient because they require shifting a potentially large number of elements. You can use a linked structure to implement a list to improve efficiency for adding and removing an element at the beginning of a list.

24.4.1 Nodes

In a linked list, each element is contained in an object, called the *node*. When a new element is added to the list, a node is created to contain it. Each node is linked to its next neighbor, as shown in Figure 24.7.

A node can be created from a class defined as follows:

```
class Node<E> {
  E element;
  Node<E> next;

  public Node(E e) {
    element = e;
  }
}
```

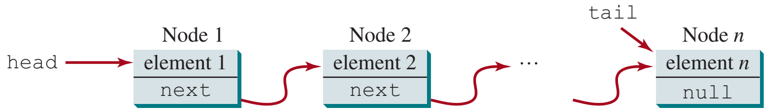

tail
Node 1
Node 2
Node *n*
head
element 1
next
element 2
next
…
element *n*
null

Figure 24.7 A linked list consists of any number of nodes chained together.

We use the variable **`head`** to refer to the first node in the list and the variable **`tail`** to the last node. If the list is empty, both **`head`** and **`tail`** are **`null`**. Here is an example that creates a linked list to hold three nodes. Each node stores a string element.

Step 1: Declare **`head`** and **`tail`**.

```
Node<String> head = null;      The list is empty now
Node<String> tail = null;
```

`head` and **`tail`** are both **`null`**. The list is empty.

Step 2: Create the first node and append it to the list, as shown in Figure 24.8. After the first node is inserted in the list, **`head`** and **`tail`** point to this node.

```
head = new Node<>("Chicago");
tail = head;
```

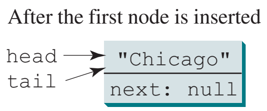
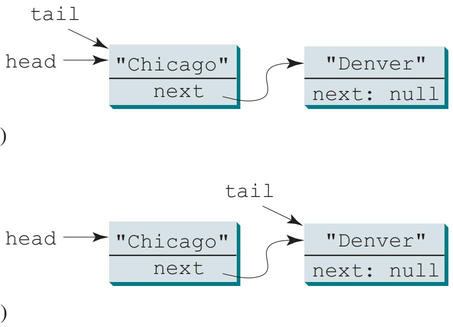

After the first node is inserted
head
tail
"Chicago"
next: null

Figure 24.8 Append the first node to the list.

Step 3: Create the second node and append it into the list, as shown in Figure 24.9a. To append the second node to the list, link the first node with the new node. The new node is now the tail node, so you should move **`tail`** to point to this new node, as shown in Figure 24.9b.

```
tail.next = new Node<>("Denver");
```

```
tail = tail.next;
```


tail
head
"Chicago"
next
"Denver"
next: null
tail
head
"Chicago"
next
"Denver"
next: null

(a)

(b)

Figure 24.9 Append the second node to the list.

Step 4: Create the third node and append it to the list, as shown in Figure 24.10a. To append the new node to the list, link the last node in the list with the new node. The new node is now the tail node, so you should move **`tail`** to point to this new node, as shown in Figure 24.10b.

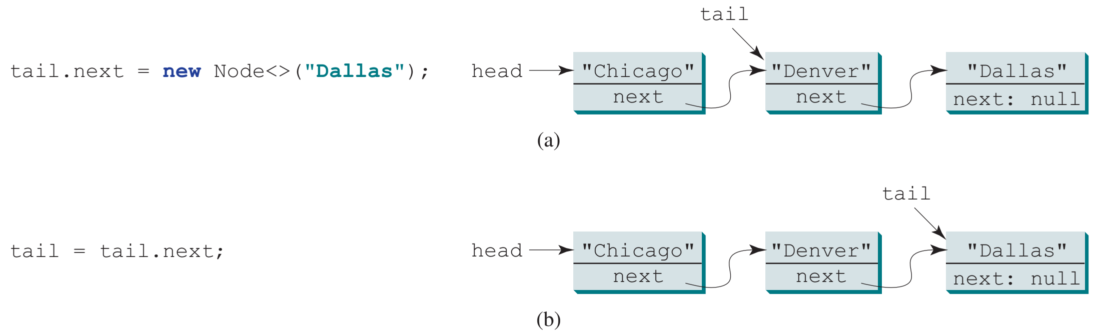

tail.next = new Node<>("Dallas");
tail
head
"Chicago"
next
"Denver"
next
"Dallas"
next: null
(a)
tail = tail.next;
tail
head
"Chicago"
next
"Denver"
next
"Dallas"
next: null
(b)

FIGURE 24.10 Append the third node to the list.

Each node contains the element and a data field named **`next`** that points to the next element. If the node is the last in the list, its pointer data field **`next`** contains the value **`null`**. You can use this property to detect the last node. For example, you can write the following loop to traverse all the nodes in the list:

```
1  Node<E> current = head;
2  while (current != null) {
3    System.out.println(current.element);
4    current = current.next;
5  }
```

current pointer
check last node
next node

The variable **`current`** points initially to the first node in the list (line 1). In the loop, the element of the current node is retrieved (line 3) then **`current`** points to the next node (line 4). The loop continues until the current node is **`null`**.

24.4.2 The `MyLinkedList` Class

The **`MyLinkedList`** class uses a linked structure to implement a dynamic list. It implements **`MyList`**. In addition, it provides the methods **`addFirst`**, **`addLast`**, **`removeFirst`**, **`removeLast`**, **`getFirst`**, and **`getLast`**, as shown in Figure 24.11.

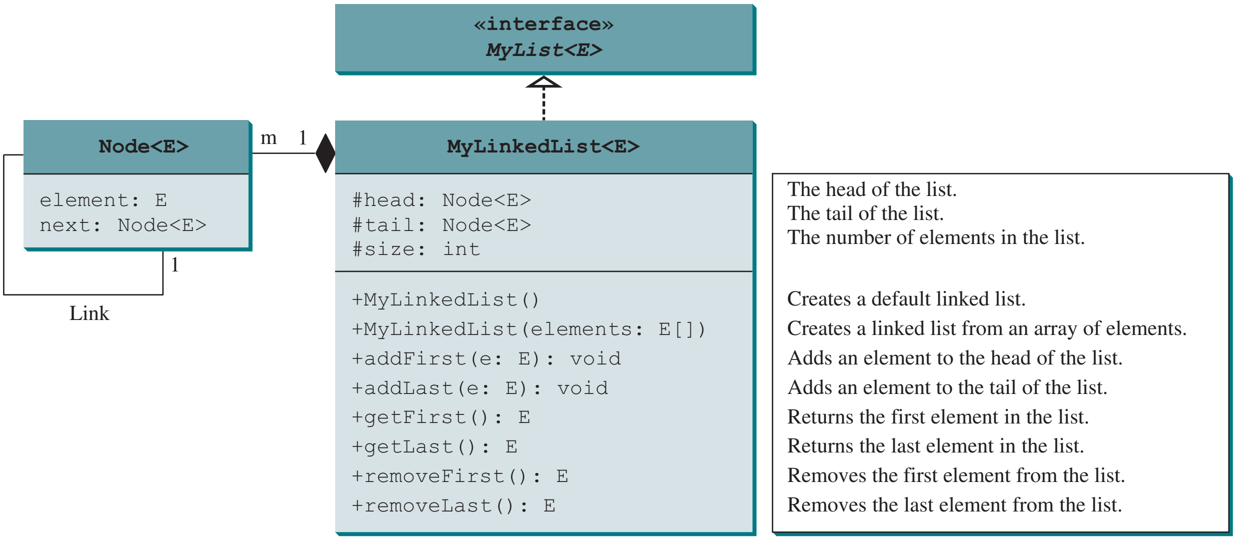

«interface»
MyList<E>
Node<E>
element: E
next: Node<E>
m
1
1
Link
MyLinkedList<E>
#head: Node<E>
#tail: Node<E>
#size: int
+MyLinkedList()
+MyLinkedList(elements: E[])
+addFirst(e: E): void
+addLast(e: E): void
+getFirst(): E
+getLast(): E
+removeFirst(): E
+removeLast(): E
The head of the list.
The tail of the list.
The number of elements in the list.
Creates a default linked list.
Creates a linked list from an array of elements.
Adds an element to the head of the list.
Adds an element to the tail of the list.
Returns the first element in the list.
Returns the last element in the list.
Removes the first element from the list.
Removes the last element from the list.

FIGURE 24.11 `MyLinkedList` implements a list using a linked list of nodes.

Assuming the class has been implemented, Listing 24.4 gives a test program that uses the class.

LISTING 24.4 TestMyLinkedList.java

create list
append element
print list
insert element
append element
append element
insert element
insert element
insert element
remove element
remove element
remove element
traverse using iterator

```java
public class TestMyLinkedList {
  /** Main method */
  public static void main(String[] args) {
    // Create a list for strings
    MyLinkedList<String> list = new MyLinkedList<>();

    // Add elements to the list
    list.add("America"); // Add it to the list
    System.out.println("(1) " + list);

    list.add(0, "Canada"); // Add it to the beginning of the list
    System.out.println("(2) " + list);

    list.add("Russia"); // Add it to the end of the list
    System.out.println("(3) " + list);

    list.addLast("France"); // Add it to the end of the list
    System.out.println("(4) " + list);

    list.add(2, "Germany"); // Add it to the list at index 2
    System.out.println("(5) " + list);

    list.add(5, "Norway"); // Add it to the list at index 5
    System.out.println("(6) " + list);

    list.add(0, "Poland"); // Same as list.addFirst("Poland")
    System.out.println("(7) " + list);

    // Remove elements from the list
    list.remove(0); // Same as list.remove("Poland") in this case
    System.out.println("(8) " + list);

    list.remove(2); // Remove the element at index 2
    System.out.println("(9) " + list);

    list.remove(list.size() - 1); // Remove the last element
    System.out.print("(10) " + list + "\n(11) ");

    for (String s: list)
      System.out.print(s.toUpperCase() + " ");

    list.clear();
    System.out.println("\nAfter clearing the list, the list size is "
      + list.size());
  }
}
```

```
(1) [America]
(2) [Canada, America]
(3) [Canada, America, Russia]
(4) [Canada, America, Russia, France]
(5) [Canada, America, Germany, Russia, France]
(6) [Canada, America, Germany, Russia, France, Norway]
(7) [Poland, Canada, America, Germany, Russia, France, Norway]
```

```
(8) [Canada, America, Germany, Russia, France, Norway]
(9) [Canada, America, Russia, France, Norway]
(10) [Canada, America, Russia, France]
(11) CANADA AMERICA RUSSIA FRANCE
After clearing the list, the list size is 0
```

24.4.3 Implementing `MyLinkedList`

Now let us turn our attention to implementing the **`MyLinkedList`** class. We will discuss how to implement the methods **`addFirst`**, **`addLast`**, **`add(index, e)`**, **`removeFirst`**, **`removeLast`**, and **`remove(index)`** and leave the other methods in the **`MyLinkedList`** class as exercises.

24.4.3.1 Implementing `addFirst(e)`

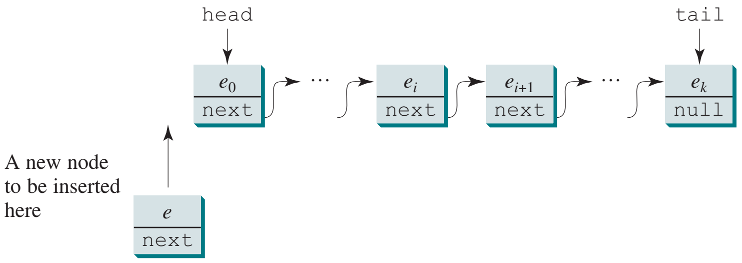
The **`addFirst(e)`** method creates a new node for holding element **`e`**. The new node becomes the first node in the list. It can be implemented as follows:

```
1  public void addFirst(E e) {
2    Node<E> newNode = new Node<>(e); // Create a new node
3    newNode.next = head; // link the new node with the head
4    head = newNode; // head points to the new node
5    size++; // Increase list size
6
7    if (tail == null) // The new node is the only node in list
8      tail = head;
9  }
```

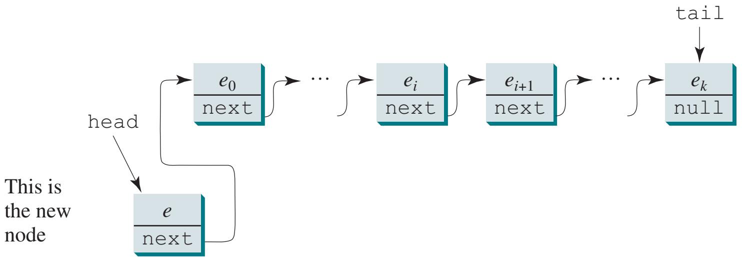
create a node
link with head
head to new node
increase size

was empty?

The **`addFirst(e)`** method creates a new node to store the element (line 2) and inserts the node at the beginning of the list (line 3), as shown in Figure 24.12a. After the insertion, **`head`** should point to this new element node (line 4), as shown in Figure 24.12b.



(a) Before a new node is inserted.



(b) After a new node is inserted.

FIGURE 24.12 A new element is added to the beginning of the list.

If the list is empty (line 7), both **`head`** and **`tail`** will point to this new node (line 8). After the node is created, **`size`** should be increased by **`1`** (line 5).

24.4.3.2 Implementing `addLast(e)`

The `addLast(e)` method creates a node to hold the element and appends the node at the end of the list. It can be implemented as follows:

create a node

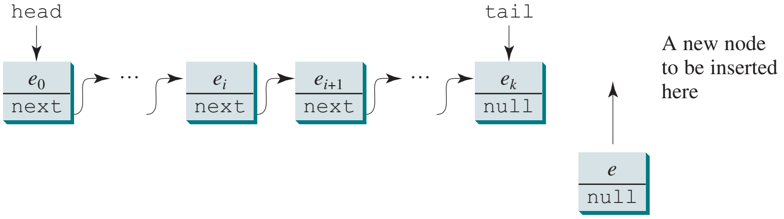
increase size

```
 1  public void addLast(E e) {
 2    Node<E> newNode = new Node<>(e); // Create a new node for e
 3
 4    if (tail == null) {
 5      head = tail = newNode; // The only node in list
 6    }
 7    else {
 8      tail.next = newNode; // Link the new node with the last node
 9      tail = newNode; // tail now points to the last node
10    }
11
12    size++; // Increase size
13  }
```

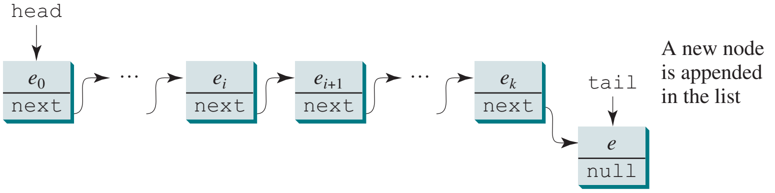
The `addLast(e)` method creates a new node to store the element (line 2) and appends it to the end of the list. Consider two cases:

1. If the list is empty (line 4), both `head` and `tail` will point to this new node (line 5);
2. Otherwise, link the node with the last node in the list (line 8). `tail` should now point to this new node (line 9). Figures 24.13a and 13b show the new node for element `e` before and after the insertion.

In any case, after the node is created, the `size` should be increased by `1` (line 12).



(a) Before a new node is inserted.



(b) After a new node is inserted.

FIGURE 24.13 A new element is added at the end of the list.

24.4.3.3 Implementing `add(index, e)`

The `add(index, e)` method inserts an element into the list at the specified index. It can be implemented as follows:

insert first
insert last

```
public void add(int index, E e) {
  if (index == 0) addFirst(e); // Insert first
  else if (index >= size) addLast(e); // Insert last
  else { // Insert in the middle
    Node<E> current = head;
```

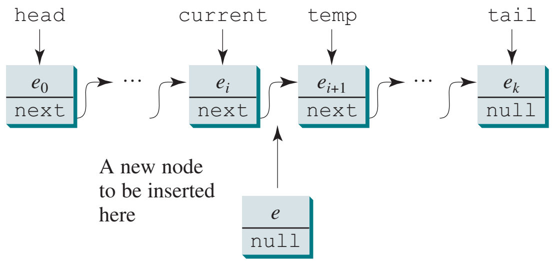
```
 6       for (int i = 1; i < index; i++)
 7         current = current.next;
 8       Node<E> temp = current.next;
 9       current.next = new Node<>(e);
10       (current.next).next = temp;
11       size++;
12     }
13   }
```

create a node

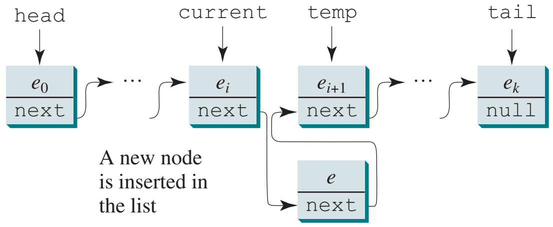
increase size

There are three cases when inserting an element into the list:

1. If `index` is `0`, invoke `addFirst(e)` (line 2) to insert the element at the beginning of the list.
2. If `index` is greater than or equal to `size`, invoke `addLast(e)` (line 3) to insert the element at the end of the list.
3. Otherwise, create a new node to store the new element and locate where to insert it. As shown in Figure 24.14a, the new node is to be inserted between the nodes `current` and `temp`. The method assigns the new node to `current.next` and assigns `temp` to the new node's `next`, as shown in Figure 24.14b. The size is now increased by `1` (line 11).


head
current
temp
tail
e_0
next
...
e_i
next
e_{i+1}
next
...
e_k
null
A new node
to be inserted
here
e
null

(a) Before a new node is inserted.


head
current
temp
tail
e_0
next
...
e_i
next
e_{i+1}
next
...
e_k
null
A new node
is inserted in
the list
e
next

(b) After a new node is inserted.

FIGURE 24.14 A new element is inserted in the middle of the list.

24.4.3.4 Implementing `removeFirst()`

The `removeFirst()` method removes the first element from the list. It can be implemented as follows:

```
public E removeFirst() {
  if (size == 0) return null; // Nothing to delete
  else {
    Node<E> temp = head; // Keep the first node temporarily
    head = head.next; // Move head to point to next node
    size--; // Reduce size by 1
    if (head == null) tail = null; // List becomes empty
```

nothing to remove

keep old head

new head

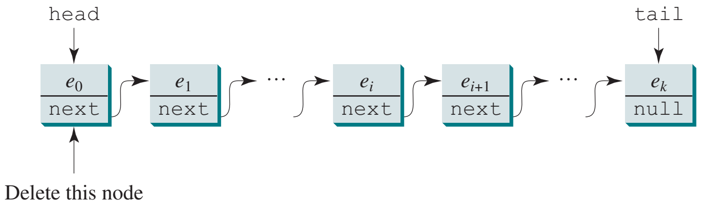
decrease size

destroy the node

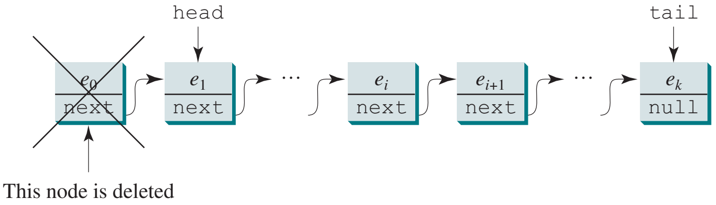
```
 8      return temp.element; // Return the deleted element
 9    }
10  }
```

Consider two cases:

1. If the list is empty, there is nothing to delete, so return `null` (line 2).
2. Otherwise, remove the first node from the list by pointing `head` to the second node. Figures 24.15a and 15b show the linked list before and after the deletion. The size is reduced by `1` after the deletion (line 6). If the list becomes empty, after removing the element, `tail` should be set to `null` (line 7).


head
tail
e_0
next
e_1
next
…
e_i
next
e_{i+1}
next
…
e_k
null
Delete this node

(a) Before the node is deleted.


head
tail
e_0
next
e_1
next
…
e_i
next
e_{i+1}
next
…
e_k
null
This node is deleted

(b) After the node is deleted.

FIGURE 24.15 The first node is deleted from the list.

24.4.3.5 Implementing `removeLast()`

The `removeLast()` method removes the last element from the list. It can be implemented as follows:

empty? or size 1?
reduce to removeFirst()

locate current before `tail`

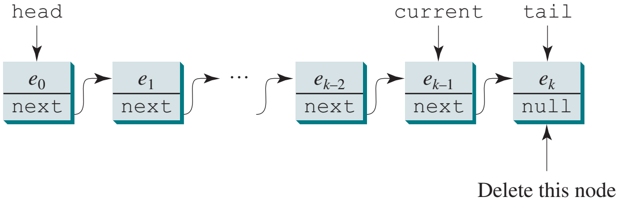
move `tail`

reduce `size`
return deleted element

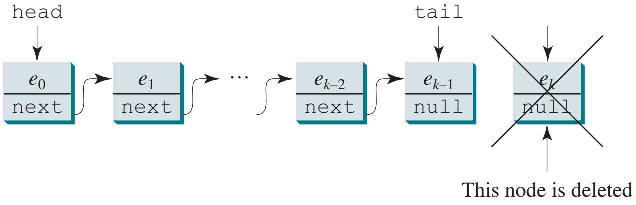
```
public E removeLast() {
  if (size == 0 || size == 1) {
     return removeFirst();
  }
  else {
    Node<E> current = head;
    for (int i = 0; i < size - 2; i++) {
      current = current.next;
    }

    E temp = tail.element;
    tail = current;
    tail.next = null;
    size--;
    return temp;
  }
}
```

Consider three cases:

1. If the list is empty or has a single element, invoking `removeFirst()` will take care of this case (lines 2–4).
2. Otherwise, locate `current` to point to the second-to-last node (lines 6–9). Save the value of tail to `temp` (line 11). Set `tail` to `current` (line 12). `tail` is now repositioned to point to the second-to-last node and destroy the last node (line 13). The size is reduced by `1` after the deletion (line 14) and the element value of the deleted node is returned (line 15).



(a) Before the node is deleted.



(b) After the node is deleted.

FIGURE 24.16 The last node is deleted from the list.

24.4.3.6 Implementing `remove(index)`

The `remove(index)` method finds the node at the specified index then removes it. It can be implemented as follows:

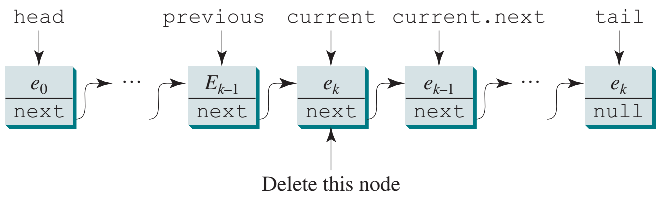
```
 1  public E remove(int index) {
 2    if (index < 0 || index >= size) return null; // Out of range
 3    else if (index == 0) return removeFirst(); // Remove first
 4    else if (index == size - 1) return removeLast(); // Remove last
 5    else {
 6      Node<E> previous = head;
 7
 8      for (int i = 1; i < index; i++) {
 9        previous = previous.next;
10      }
11
12      Node<E> current = previous.next;
13      previous.next = current.next;
14      size--;
15      return current.element;
16    }
17  }
```

out of range
remove first
remove last
locate previous
locate current
remove from list
reduce size
return element

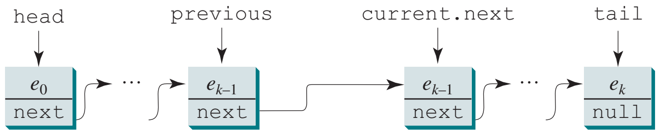
Consider four cases:

1. If `index` is beyond the range of the list (i.e., `index < 0 || index >= size`), return `null` (line 2).

2. If `index` is `0`, invoke `removeFirst()` to remove the first node (line 3).
3. If `index` is `size – 1`, invoke `removeLast()` to remove the last node (line 4).
4. Otherwise, locate the node at the specified `index`. Let `current` denote this node and `previous` denote the node before this node, as shown in Figure 24.17a. Assign `current.next` to `previous.next` to eliminate the current node, as shown in Figure 24.17b.



(a) Before the node is deleted.



(b) After the node is deleted.

FIGURE 24.17 A node is deleted from the list.

iterator

Listing 24.5 gives the implementation of `MyLinkedList`. The implementation of `get(index)`, `indexOf(e)`, `lastIndexOf(e)`, `contains(e)`, and `set(index, e)` is omitted and left as an exercise. The `iterator()` method defined in the `java.lang.Iterable` interface is implemented to return an instance on `java.util.Iterator` (lines 128–130). The `LinkedListIterator` class implements `Iterator` with concrete methods for `hasNext`, `next`, and `remove` (lines 132–152). This implementation uses `current` to point to the current position of the element being traversed (line 134). Initially, `current` points to the head of the list.

LISTING 24.5 MyLinkedList.java

head, tail
number of elements
no-arg constructor
constructor
getFirst

```
 1  public class MyLinkedList<E> implements MyList<E> {
 2    private Node<E> head, tail;
 3    private int size = 0; // Number of elements in the list
 4
 5    /** Create an empty list */
 6    public MyLinkedList() {
 7    }
 8
 9    /** Create a list from an array of objects */
10    public MyLinkedList(E[] objects) {
11      for (int i = 0; i < objects.length; i++)
12        add(objects[i]);
13    }
14
15    /** Return the head element in the list */
16    public E getFirst() {
17      if (size == 0) {
18        return null;
19      }
20      else {
      return head.element;
    }
  }

  /** Return the last element in the list */
```

getLast

```
  public E getLast() {
    if (size == 0) {
      return null;
    }
    else {
      return tail.element;
    }
  }

  /** Add an element to the beginning of the list */
```

addFirst

```
  public void addFirst(E e) {
    // Implemented in Section 24.4.3.1, so omitted here
  }

  /** Add an element to the end of the list */
```

addLast

```
  public void addLast(E e) {
    // Implemented in Section 24.4.3.2, so omitted here
  }

  @Override /** Add a new element at the specified index
   * in this list. The index of the head element is 0 */
```

add

```
  public void add(int index, E e) {
    // Implemented in Section 24.4.3.3, so omitted here
  }

  /** Remove the head node and
   * return the object that is contained in the removed node. */
```

removeFirst

```
  public E removeFirst() {
    // Implemented in Section 24.4.3.4, so omitted here
  }

  /** Remove the last node and
   * return the object that is contained in the removed node. */
```

removeLast

```
  public E removeLast() {
    // Implemented in Section 24.4.3.5, so omitted here
  }

  @Override /** Remove the element at the specified position in this
   * list. Return the element that was removed from the list. */
```

remove

```
  public E remove(int index) {
    // Implemented earlier in Section 24.4.3.6, so omitted
  }

  @Override /** Override toString() to return elements in the list */
```

toString

```
  public String toString() {
    StringBuilder result = new StringBuilder("[");

    Node<E> current = head;
    for (int i = 0; i < size; i++) {
      result.append(current.element);
      current = current.next;
      if (current != null) {
        result.append(", "); // Separate two elements with a comma
      }
      else {
        result.append("]"); // Insert the closing ] in the string
      }
    }

    return result.toString();
  }

  @Override /** Clear the list */
```

clear

```
  public void clear() {
    size = 0;
    head = tail = null;
  }

  @Override /** Return true if this list contains the element e */
```

contains

```
  public boolean contains(Object e) {
    // Left as an exercise
    return true;
  }

  @Override /** Return the element at the specified index */
```

get

```
  public E get(int index) {
    // Left as an exercise
    return null;
  }

  @Override /** Return the index of the first matching element in
   * this list. Return -1 if no match. */
```

indexOf

```
  public int indexOf(Object e) {
    // Left as an exercise
    return 0;
  }

  @Override /** Return the index of the last matching element in
   * this list. Return -1 if no match. */
```

lastIndexOf

```
  public int lastIndexOf(E e) {
    // Left as an exercise
    return 0;
  }

  @Override /** Replace the element at the specified position
   * in this list with the specified element. */
```

set

```
  public E set(int index, E e) {
    // Left as an exercise
    return null;
  }

  @Override /** Override iterator() defined in Iterable */
```

iterator

```
  public java.util.Iterator<E> iterator() {
    return new LinkedListIterator();
  }

```

LinkedListIterator class

```
  private class LinkedListIterator
      implements java.util.Iterator<E> {
    private Node<E> current = head; // Current index

    @Override
    public boolean hasNext() {
      return (current != null);
    }

    @Override
    public E next() {
      E e = current.element;
      current = current.next;
      return e;
    }

    @Override
    public void remove() {
      // Left as an exercise
    }
  }

  private static class Node<E> {
    E element;
    Node<E> next;

    public Node(E element) {
      this.element = element;
    }
  }

  @Override /** Return the number of elements in this list */
  public int size() {
    return size;
  }
}
```

Node inner class

24.4.4 `MyArrayList` vs. `MyLinkedList`

Both `MyArrayList` and `MyLinkedList` can be used to store a list. `MyArrayList` is implemented using an array, and `MyLinkedList` is implemented using a linked list. The overhead of `MyArrayList` is smaller than that of `MyLinkedList`. However, `MyLinkedList` is more efficient if you need to insert elements into and delete elements from the beginning of the list. Table 24.1 summarizes the complexity of the methods in `MyArrayList` and `MyLinkedList`. Note `MyArrayList` is the same as `java.util.ArrayList`, and `MyLinkedList` is the same as `java.util.LinkedList` except that `MyLinkedList` is implemented using singly a linked list and `LinkedList` is implemented using a doubly linked list. We will introduce doubly linked lists in Section 24.4.5.

TABLE 24.1 Time Complexities for Methods in `MyArrayList` and `MyLinkedList`

Methods	*MyArrayList*	*MyLinkedList*
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`get(index: int)`	`O(1)`	`O(n)`
`indexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`size()`	$O(1)$	$O(1)$
`remove(index: int)`	$O(n)$	$O(n)$
`set(index: int, e: E)`	$O(n)$	$O(n)$
`addFirst(e: E)`	`O(n)`	`O(1)`
`removeFirst()`	`O(n)`	`O(1)`

Note that you can implement `MyLinkedList` without using the `size` data field. But then the `size()` method would take `O(n)` time.

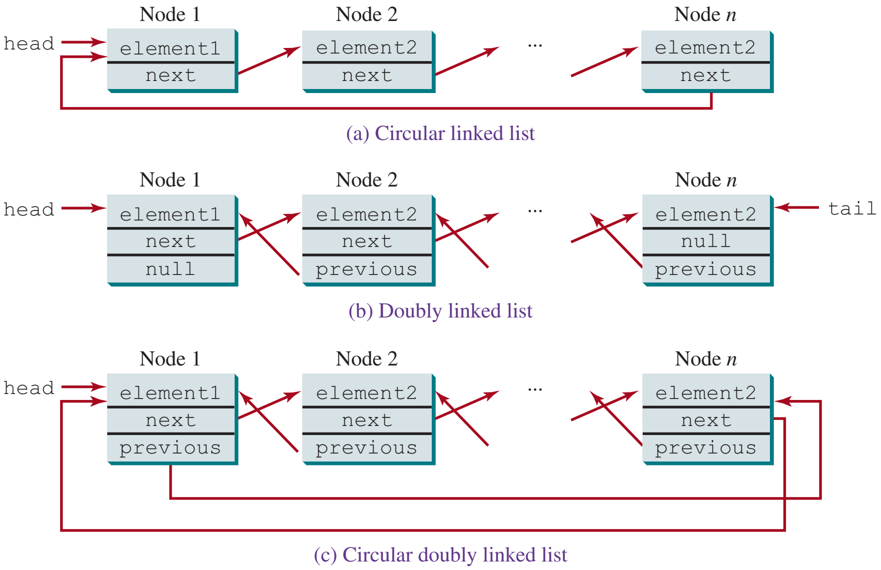
Note

The `MyArrayList` is implemented using an array. In a `MyLinkedList`, every element is wrapped in an object. `MyArrayList` has less overhead than `MyLinkedList`. You should use `MyLinkedList` only if the application involves frequently inserting/deleting the elements from the beginning of a list.

24.4.5 Variations of Linked Lists

The linked list introduced in the preceding sections is known as a *singly linked list*. It contains a pointer to the list's first node, and each node contains a pointer to the next node sequentially. Several variations of the linked list are useful in certain applications.

A *circular, singly linked list* is like a singly linked list, except that the pointer of the last node points back to the first node, as shown in Figure 24.18a. Note `tail` is not needed for circular linked lists. `head` points to the current node in the list. Insertion and deletion take place at the current node. A good application of a circular linked list is in the operating system that serves multiple users in a timesharing fashion. The system picks a user from a circular list and grants a small amount of CPU time then moves on to the next user in the list.



(a) Circular linked list

(b) Doubly linked list

(c) Circular doubly linked list

FIGURE 24.18 Linked lists may appear in various forms.

A *doubly linked list* contains nodes with two pointers. One points to the next node and the other to the previous node, as shown in Figure 24.18b. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. Thus, a doubly linked list can be traversed forward and backward. The `java.util.LinkedList` class is implemented using a doubly linked list, and it supports traversing of the list forward and backward using the `ListIterator`.

A *circular*, *doubly linked list* is like a doubly linked list, except that the forward pointer of the last node points to the first node, and the backward pointer of the first pointer points to the last node, as shown in Figure 24.18c.

The implementations of these linked lists are left as exercises.

Note

In a singly linked list, `removeLast()` takes O(n) time. In a doubly linked list, `removeLast()` can be implemented to take O(1) time. The `LinkedList` in the Java API is implemented using a doubly linked list. See CheckPoint 24.4.11.

Check Point

24.4.1 If a linked list does not contain any nodes, what are the values in `head` and `tail`?

24.4.2 If a linked list has only one node, is `head == tail` true? List all cases in which `head == tail` is true.

24.4.3 Draw a diagram to show the linked list after each of the following statements is executed:

```
MyLinkedList<Double> list = new MyLinkedList<>();
list.add(1.5);
list.add(6.2);
list.add(3.4);
list.add(7.4);
list.remove(1.5);
list.remove(2);
```

24.4.4 When a new node is inserted to the head of a linked list, will the `head` and the `tail` be changed?

24.4.5 When a new node is appended to the end of a linked list, will the `head` and the `tail` be changed?

24.4.6 Simplify the code in lines 77–82 in Listing 24.5 using a conditional expression.

24.4.7 What is the time complexity of the `addFirst(e)` and `removeFirst()` methods in `MyLinkedList`?

24.4.8 Suppose you need to store a list of elements. If the number of elements in the program is fixed, what data structure should you use? If the number of elements in the program changes, what data structure should you use?

24.4.9 If you have to add or delete the elements at the beginning of a list, should you use `MyArrayList` or `MyLinkedList`? If most of the operations on a list involve retrieving an element at a given index, should you use `MyArrayList` or `MyLinkedList`?

24.4.10 Both `MyArrayList` and `MyLinkedList` are used to store a list of objects. Why do we need both types of lists?

24.4.11 Implement the `removeLast()` method in a doubly linked list in O(1) time.

24.5 Stacks and Queues

Stacks can be implemented using array lists and queues can be implemented using linked lists.

Key Point

A stack can be viewed as a special type of list whose elements are accessed, inserted, and deleted only from the end (top), as shown in Figure 10.11. A queue represents a waiting list. It can be viewed as a special type of list whose elements are inserted into the end (tail) of the queue and are accessed and deleted from the beginning (head), as shown in Figure 24.19.

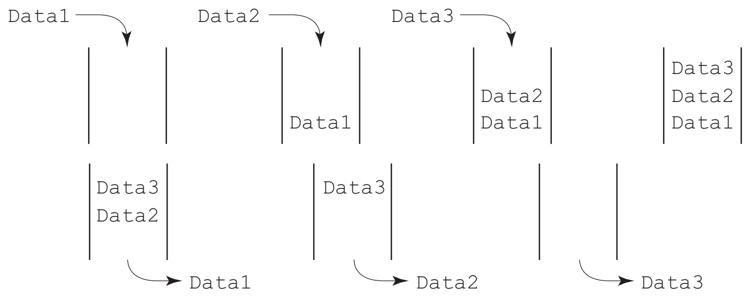
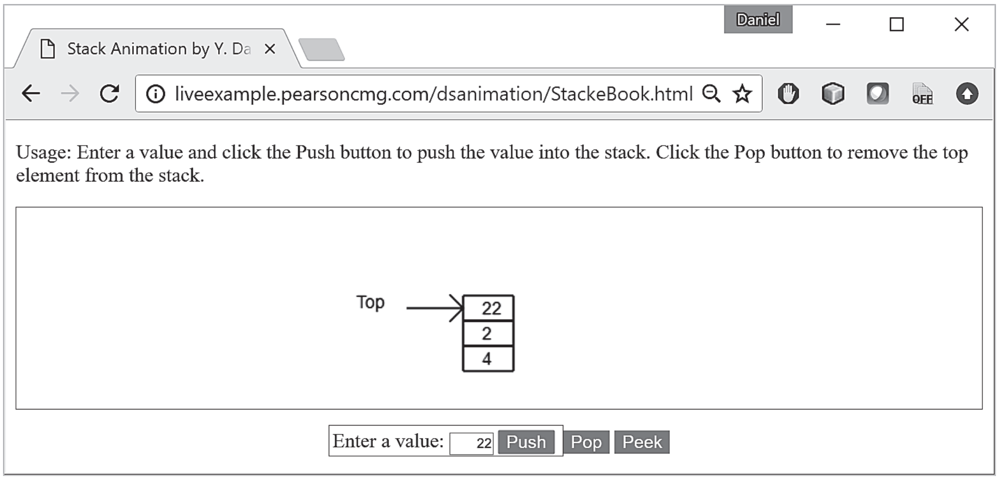


FIGURE 24.19 A queue holds objects in a first-in, first-out fashion.

stack and queue animation on Companion Website

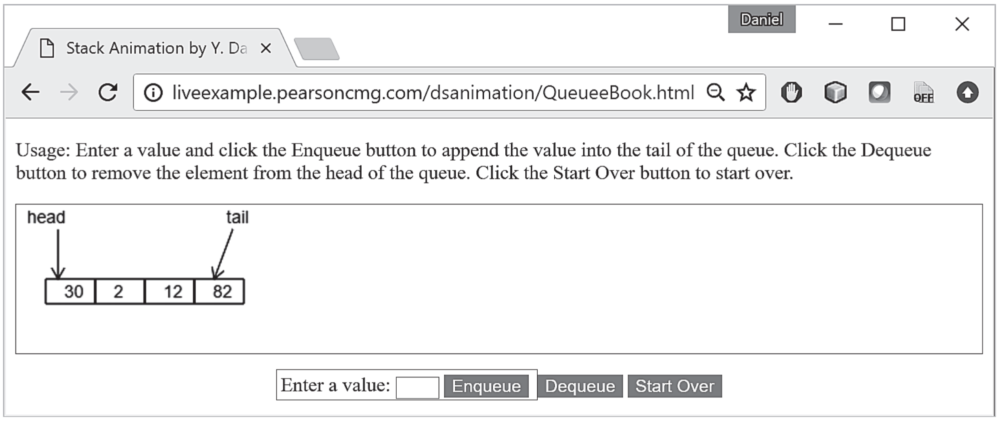
Pedagogical Note

For an interactive demo on how stacks and queues work, go to liveexample.pearsoncmg.com/dsanimation/StackeBook.html, and liveexample.pearsoncmg.com/dsanimation/QueueeBook.html, as shown in Figure 24.20.


Daniel
Stack Animation by Y. Da
liveexample.pearsoncmg.com/dsanimation/StackeBook.html
Top
22
2
4
Enter a value:
22
Push
Pop
Peek

Usage: Enter a value and click the Push button to push the value into the stack. Click the Pop button to remove the top element from the stack.

(a) Stack animation


Daniel
Stack Animation by Y. Da
liveexample.pearsoncmg.com/dsanimation/QueueeBook.html
head
tail
30
2
12
82
Enter a value:
Enqueue
Dequeue
Start Over

Usage: Enter a value and click the Enqueue button to append the value into the tail of the queue. Click the Dequeue button to remove the element from the head of the queue. Click the Start Over button to start over.

(b) Queue animation

FIGURE 24.20 The animation tool enables you to see how queues work. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Since the insertion and deletion operations on a stack are made only at the end of the stack, it is more efficient to implement a stack with an array list than a linked list. Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list. This section implements a stack class using an array list and a queue class using a linked list.

There are two ways to design the stack and queue classes:

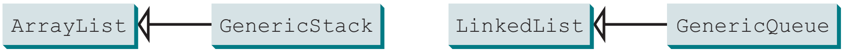
inheritance

- Using inheritance: You can define a stack class by extending `ArrayList`, and a queue class by extending `LinkedList`, as shown in Figure 24.21a.

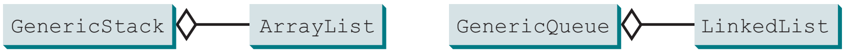
composition

- Using composition: You can define an array list as a data field in the stack class and a linked list as a data field in the queue class, as shown in Figure 24.21b.

Both designs are fine, but using composition is better because it enables you to define a completely new stack class and queue class without inheriting the unnecessary and inappropriate methods from the array list and linked list. The implementation of the stack class using the


ArrayList
GenericStack
LinkedList
GenericQueue

(a) Using inheritance


GenericStack
ArrayList
GenericQueue
LinkedList

(b) Using composition

FIGURE 24.21 `GenericStack` and `GenericQueue` may be implemented using inheritance or composition.

composition approach was given in Listing 19.1, GenericStack.java. Listing 24.6 implements the `GenericQueue` class using the composition approach. Figure 24.22 shows the UML of the class.

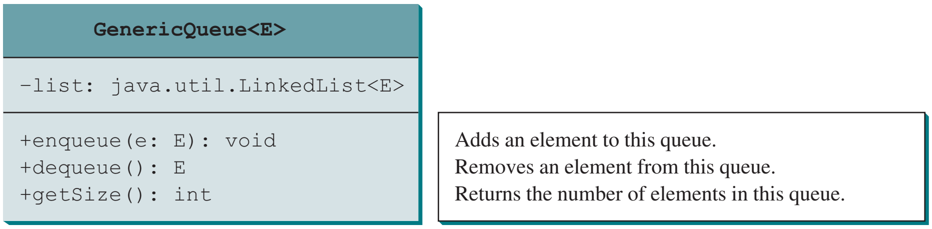

GenericQueue<E>
-list: java.util.LinkedList<E>
+enqueue(e: E): void
+dequeue(): E
+getSize(): int
Adds an element to this queue.
Removes an element from this queue.
Returns the number of elements in this queue.

FIGURE 24.22 `GenericQueue` uses a linked list to provide a first-in, first-out data structure.

LISTING 24.6 GenericQueue.java

```
 1  public class GenericQueue<E> {
 2    private java.util.LinkedList<E> list
 3      = new java.util.LinkedList<>();
 4
 5    public void enqueue(E e) {
 6      list.addLast(e);
 7    }
 8
 9    public E dequeue() {
10      return list.removeFirst();
11    }
12
13    public int getSize() {
14      return list.size();
15    }
16
17    @Override
18    public String toString() {
19      return "Queue: " + list.toString();
20    }
21  }
```

linked list

`enqueue`

`dequeue`

`getSize`

`toString`

A linked list is created to store the elements in a queue (lines 2 and 3). The `enqueue(e)` method (lines 5–7) adds element `e` into the tail of the queue. The `dequeue()` method (lines 9–11) removes an element from the head of the queue and returns the removed element. The `getSize()` method (lines 13–15) returns the number of elements in the queue.

Listing 24.7 gives an example that creates a stack using `GenericStack` and a queue using `GenericQueue`. It uses the `push` (`enqueue`) method to add strings to the stack (queue) and the `pop` (`dequeue`) method to remove strings from the stack (queue).

LISTING 24.7 TestStackQueue.java

```
public class TestStackQueue {
  public static void main(String[] args) {
    // Create a stack
    GenericStack<String> stack = new GenericStack<>();

    // Add elements to the stack
    stack.push("Tom"); // Push Tom to the stack
    System.out.println("(1) " + stack);

    stack.push("Susan"); // Push Susan to the the stack
    System.out.println("(2) " + stack);

    stack.push("Kim"); // Push Kim to the stack
    stack.push("Michael"); // Push Michael to the stack
    System.out.println("(3) " + stack);

    // Remove elements from the stack
    System.out.println("(4) " + stack.pop());
    System.out.println("(5) " + stack.pop());
    System.out.println("(6) " + stack);

    // Create a queue
    GenericQueue<String> queue = new GenericQueue<>();

    // Add elements to the queue
    queue.enqueue("Tom"); // Add Tom to the queue
    System.out.println("(7) " + queue);

    queue.enqueue("Susan"); // Add Susan to the queue
    System.out.println("(8) " + queue);

    queue.enqueue("Kim"); // Add Kim to the queue
    queue.enqueue("Michael"); // Add Michael to the queue
    System.out.println("(9) " + queue);

    // Remove elements from the queue
    System.out.println("(10) " + queue.dequeue());
    System.out.println("(11) " + queue.dequeue());
    System.out.println("(12) " + queue);
  }
}
```

```
(1) stack: [Tom]
(2) stack: [Tom, Susan]
(3) stack: [Tom, Susan, Kim, Michael]
(4) Michael
(5) Kim
(6) stack: [Tom, Susan]
(7) Queue: [Tom]
(8) Queue: [Tom, Susan]
(9) Queue: [Tom, Susan, Kim, Michael]
(10) Tom
(11) Susan
(12) Queue: [Kim, Michael]
```

stack time complexity

For a stack, the **push(e)** method adds an element to the top of the stack, and the **pop()** method removes the top element from the stack and returns the removed element. It is easy to see that the time complexity for the **push** and **pop** methods is $O(1)$.

queue time complexity

For a queue, the `enqueue(e)` method adds an element to the tail of the queue, and the `dequeue()` method removes the element from the head of the queue. It is easy to see that the time complexity for the `enqueue` and `dequeue` methods is *O*(1).

24.5.1 You can use inheritance or composition to design the data structures for stacks and queues. Discuss the pros and cons of these two approaches.

24.5.2 If `LinkedList` is replaced by `ArrayList` in lines 2 and 3 in Listing 24.6, GenericQueue.java, what will be the time complexity for the `enqueue` and `dequeue` methods.

24.5.3 Which lines of the following code are wrong?

```
1  List<String> list = new ArrayList<>();
2  list.add("Tom");
3  list = new LinkedList<>();
4  list.add("Tom");
5  list = new GenericStack<>();
6  list.add("Tom");
```

24.6 Priority Queues

Priority queues can be implemented using heaps.

An ordinary queue is a first-in, first-out data structure. Elements are appended to the end of the queue and removed from the beginning. In a *priority queue*, elements are assigned with priorities. When accessing elements, the element with the highest priority is removed first. For example, the emergency room in a hospital assigns priority numbers to patients; the patient with the highest priority is treated first.

A priority queue can be implemented using a heap, in which the root is the object with the highest priority in the queue. Heaps were introduced in Section 23.6, Heap Sort. The class diagram for the priority queue is shown in Figure 24.23. Its implementation is given in Listing 24.8.

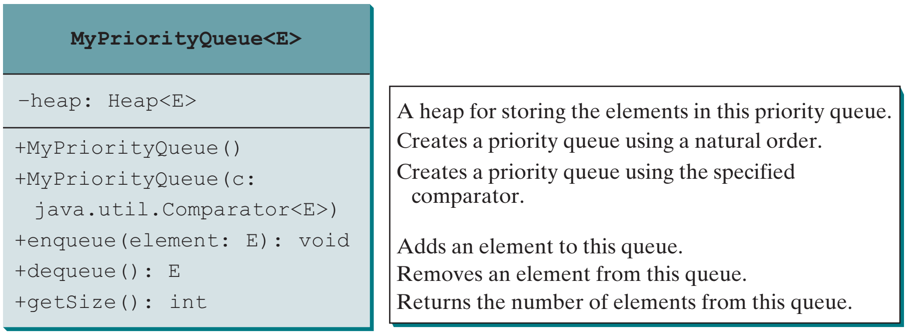

MyPriorityQueue<E>
-heap: Heap<E>
+MyPriorityQueue()
+MyPriorityQueue(c: java.util.Comparator<E>)
+enqueue(element: E): void
+dequeue(): E
+getSize(): int
A heap for storing the elements in this priority queue.
Creates a priority queue using a natural order.
Creates a priority queue using the specified comparator.
Adds an element to this queue.
Removes an element from this queue.
Returns the number of elements from this queue.

Figure 24.23 `MyPriorityQueue` uses a heap to store the elements.

Listing 24.8 MyPriorityQueue.java

```
public class MyPriorityQueue<E> {
  private Heap<E> heap;

  public void MyPriorityQueue<E> {
    heap.add(new Heap<E>();
  }

  public MyPriorityQueue(java.util.Comparator<E> c) {
    heap = new Heap<E>(c);
  }

  public void enqueue(E newObject) {
```

heap for priority queue

`no-arg constructor`

`constructor`

`enqueue`

dequeue

getsize

```
    heap.add(newObject);
  }

  public E dequeue() {
    return heap.remove();
  }

  public int getSize() {
    return heap.getSize();
  }
}
```

Listing 24.9 gives an example of using a priority queue for patients. The **`Patient`** class is defined in lines 19–37. Four patients are created with associated priority values in lines 3–6. Line 8 creates a priority queue. The patients are enqueued in lines 10–13. Line 16 dequeues a patient from the queue.

LISTING 24.9 TestPriorityQueue.java

create a patient

create a priority queue

add to queue

remove from queue

inner class Patient

compareTo

```
 1 public class TestPriorityQueue {
 2   public static void main(String[] args) {
 3     Patient patient1 = new Patient("John", 2);
 4     Patient patient2 = new Patient("Jim", 1);
 5     Patient patient3 = new Patient("Tim", 5);
 6     Patient patient4 = new Patient("Cindy", 7);
 7 
 8     MyPriorityQueue<Patient> priorityQueue
 9       = new MyPriorityQueue<>();
10     priorityQueue.enqueue(patient1);
11     priorityQueue.enqueue(patient2);
12     priorityQueue.enqueue(patient3);
13     priorityQueue.enqueue(patient4);
14 
15     while (priorityQueue.getSize() > 0)
16       System.out.print(priorityQueue.dequeue() + " ");
17   }
18 
19   static class Patient implements Comparable<Patient> {
20     private String name;
21     private int priority;
22 
23     public Patient(String name, int priority) {
24       this.name = name;
25       this.priority = priority;
26     }
27 
28     @Override
29     public String toString() {
30       return name + "(priority:" + priority + ")";
31     }
32 
33     @Override
34     public int compareTo(Patient patient) {
35       return this.priority - patient.priority;
36     }
37   }
38 }
```

```
Cindy(priority:7) Tim(priority:5) John(priority:2) Jim(priority:1)
```

Check Point

24.6.1 What is a priority queue?

24.6.2 What are the time complexity of the `enqueue`, `dequeue`, and `getSize` methods in `MyPriorityQueue`?

24.6.3 Which of the following statements are wrong?

```
1  MyPriorityQueue<Object> q1 = new MyPriorityQueue<>();
2  MyPriorityQueue<Number> q2 = new MyPriorityQueue<>();
3  MyPriorityQueue<Integer> q3 = new MyPriorityQueue<>();
4  MyPriorityQueue<Date> q4 = new MyPriorityQueue<>();
5  MyPriorityQueue<String> q5 = new MyPriorityQueue<>();
```

CHAPTER SUMMARY

1. You learned how to implement array lists, linked lists, stacks, and queues.
2. To define a data structure is essentially to define a class. The class for a data structure should use data fields to store data and provide methods to support operations such as insertion and deletion.
3. To create a data structure is to create an instance from the class. You can then apply the methods on the instance to manipulate the data structure, such as inserting an element into the data structure or deleting an element from the data structure.
4. You learned how to implement a priority queue using a heap.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

24.1 (*Implement set operations in `MyList`*) The implementations of the methods `addAll`, `removeAll`, `retainAll`, `toArray()`, and `toArray(T[])` are omitted in the `MyList` interface. Implement these methods. Test your new `MyList` class using the code at https://liveexample.pearsoncmg.com/test/Exercise24_01.txt.

***24.2** (*Implement `MyLinkedList`*) The implementations of the methods `contains(E e)`, `get(int index)`, `indexOf(E e)`, `lastIndexOf(E e)`, and `set(int index, E e)` are omitted in the `MyLinkedList` class. Implement these methods. Define a new class named `MyLinkedListExtra` that extends `MyLinkedList` to override these methods. Test your new `MyList` class using the code at https://liveexample.pearsoncmg.com/test/Exercise24_02.txt.

***24.3** (*Implement a doubly linked list*) The `MyLinkedList` class used in Listing 24.5 is a one-way directional linked list that enables one-way traversal of the list. Modify the `Node` class to add the new data field name `previous` to refer to the previous node in the list, as follows:

```
public class Node<E> {
  E element;
  Node<E> next;
  Node<E> previous;

  public Node(E e) {
    element = e;
  }
}
```

Implement a new class named **`TwoWayLinkedList`** that uses a doubly linked list to store elements. Define **`TwoWayLinkedList`** to implements **`MyList`**. You need to implement all the methods defined in **`MyLinkedList`** as well as the methods **`listIterator()`** and **`listIterator(int index)`**. Both return an instance of **`java.util.ListIterator<E>`** (see Figure 20.4). The former sets the cursor to the head of the list and the latter to the element at the specified index. Test your new class using this code from https://liveexample.pearsoncmg.com/test/Exercise24_03.txt.

24.4 (*Use the **`GenericStack`** class*) Write a program that displays the first 50 prime numbers in descending order. Use a stack to store the prime numbers.

24.5 (*Implement **`GenericQueue`** using inheritance*) In Section 24.5, Stacks and Queues, **`GenericQueue`** is implemented using composition. Define a new queue class that extends **`java.util.LinkedList`**.

****24.6** (*Revise **`MyPriorityQueue`***) Listing 24.8, uses a heap to implement the priority queue. Revise the implementation using a sorted array list to store the elements and name the new class **`PriorityQueueUsingSortedArrayList`**. The elements in the array list are sorted in increasing order of their priority with the last element having the highest priority. Write a test program that generates 5 million integers and enqueues them to the priority and dequeues from the queue. Use the same numbers for **`MyPriorityQueue`** and **`PriorityQueueUsingSortedArraList`** and display their execution times.

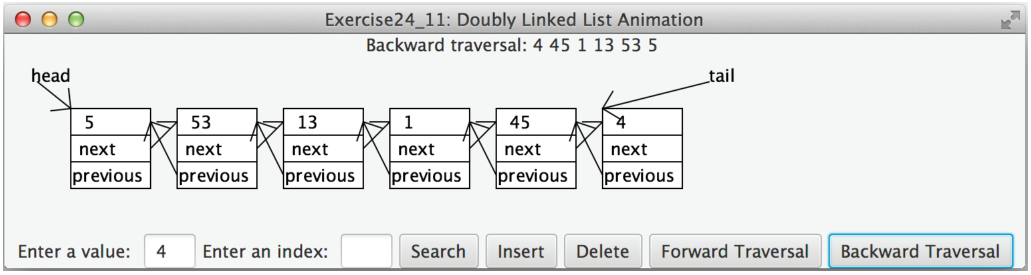
****24.7** (*Animation: linked list*) Write a program to animate search, insertion, and deletion in a linked list, as shown in Figure 24.1b. The *Search* button searches the specified value in the list. The *Delete* button deletes the specified value from the list. The *Insert* button appends the value into the list if the index is not specified; otherwise, it inserts the value into the specified index in the list.

***24.8** (*Animation: array list*) Write a program to animate search, insertion, and deletion in an array list, as shown in Figure 24.1a. The *Search* button searches the specified value in the list. The *Delete* button deletes the specified value from the list. The *Insert* button appends the value into the list if the index is not specified; otherwise, it inserts the value into the specified index in the list.

***24.9** (*Animation: array list in slow motion*) Improve the animation in the preceding programming exercise by showing the insertion and deletion operations in a slow motion.

***24.10** (*Animation: stack*) Write a program to animate push and pop in a stack, as shown in Figure 24.20a.

***24.11** (*Animation: doubly linked list*) Write a program to animate search, insertion, and deletion in a doubly linked list, as shown in Figure 24.24. The *Search* button searches the specified value in the list. The *Delete* button deletes the specified value from the list. The *Insert* button appends the value into the list if the index is not specified; otherwise, it inserts the value into the specified index in the list. Also add two buttons named *Forward Traversal* and *Backward Traversal* for displaying the elements in a forward and backward order, respectively, using iterators, as shown in Figure 24.24. The elements are displayed in a label.


Exercise24_11: Doubly Linked List Animation
head
tail
5
53
13
1
45
4
next
previous
Enter a value:
4
Enter an index:
Search
Insert
Delete
Forward Traversal
Backward Traversal

Backward traversal: 4 45 1 13 53 5

Figure 24.24 The program animates the work of a doubly linked list. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***24.12** (*Animation: queue*) Write a program to animate the `enqueue` and `dequeue` operations on a queue, as shown in Figure 24.20b.

***24.13** (*Fibonacci number iterator*) Define an iterator class named `FibonacciIterator` for iterating Fibonacci numbers. The constructor takes an argument that specifies the limit of the maximum Fibonacci number. For example, new `FibonacciIterator(23302)` creates an iterator that iterates Fibonacci numbers less than or equal to `23302`. Write a test program that uses this iterator to display all Fibonacci numbers less than or equal to `100000`.

***24.14** (*Prime number iterator*) Define an iterator class named `PrimeIterator` for iterating prime numbers. The constructor takes an argument that specifies the limit of the maximum prime number. For example, new `PrimeIterator(23302)` creates an iterator that iterates prime numbers less than or equal to `23302`. Write a test program that uses this iterator to display all prime numbers less than or equal to `100000`.

****24.15** (*Test* `MyArrayList`) Design and write a complete test program to test if the `MyArrayList` class in Listing 24.2 meets all requirements.

****24.16** (*Test* `MyLinkedList`) Design and write a complete test program to test if the `MyLinkedList` class in Listing 24.5 meets all requirements.

CHAPTER

25

BINARY SEARCH TREES

Objectives

- To design and implement a binary search tree (§25.2).
- To represent binary trees using linked data structures (§25.3).
- To search an element in a binary search tree (§25.4).
- To insert an element into a binary search tree (§25.5).
- To traverse elements in a binary tree (§25.6).
- To design and implement the `Tree` interface and the `BST` class (§25.7).
- To delete elements from a binary search tree (§25.8).
- To display a binary tree graphically (§25.9).
- To create iterators for traversing a binary tree (§25.10).
- To implement Huffman coding for compressing data using a binary tree (§25.11).

25.1 Introduction

Key Point

A binary search tree is more efficient than a list for search, insertion, and deletion operations.

The preceding chapter gives the implementation for array lists and linked lists. The time complexity of search, insertion, and deletion operations in these data structures is $O(n)$. This chapter presents a new data structure called binary search tree, which takes $O(\log n)$ average time for search, insertion, and deletion of elements.

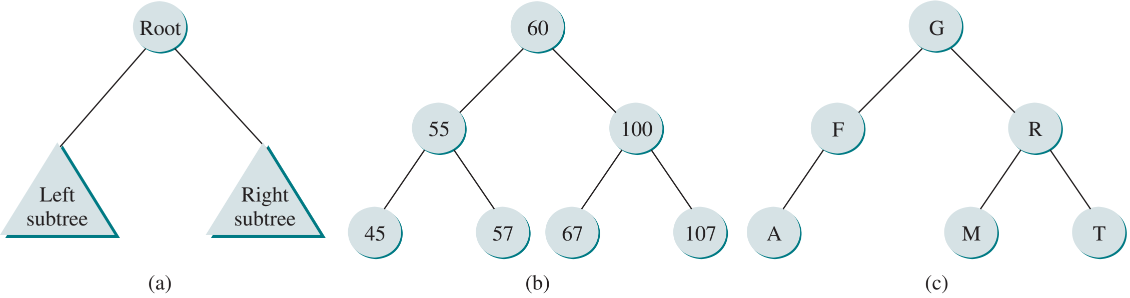
25.2 Binary Search Trees Basics

Key Point

For every node in a binary search tree, the value of its left child is less than the value of the node, and the value of its right child is greater than the value of the node.

binary tree
root
left subtree
right subtree

Recall that lists, stacks, and queues are linear structures that consist of a sequence of elements. A *binary tree* is a hierarchical structure. It either is empty or consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*, either or both of which may be empty, as shown in Figure 25.1a. Examples of binary trees are shown in Figures 25.1a and b.



(a) (b) (c)

FIGURE 25.1 Each node in a binary tree has zero, one, or two subtrees.

length
depth
level
sibling
leaf
height

The *length* of a path is the number of the edges in the path. The *depth* of a node is the length of the path from the root to the node. The set of all nodes at a given depth is sometimes called a *level* of the tree. *Siblings* are nodes that share the same parent node. The root of a left (right) subtree of a node is called a *left (right) child* of the node. A node without children is called a *leaf*. The height of a nonempty tree is the length of the path from the root node to its furthest leaf. The *height* of a tree that contains a single node is **0**. Conventionally, the height of an empty tree is **−1**. Consider the tree in Figure 25.1b. The length of the path from node 60 to 45 is **2**. The depth of node 60 is **0**, the depth of node 55 is **1**, and the depth of node 45 is **2**. The height of the tree is **2**. Nodes 45 and 57 are siblings. Nodes 45, 57, 67, and 107 are at the same level.

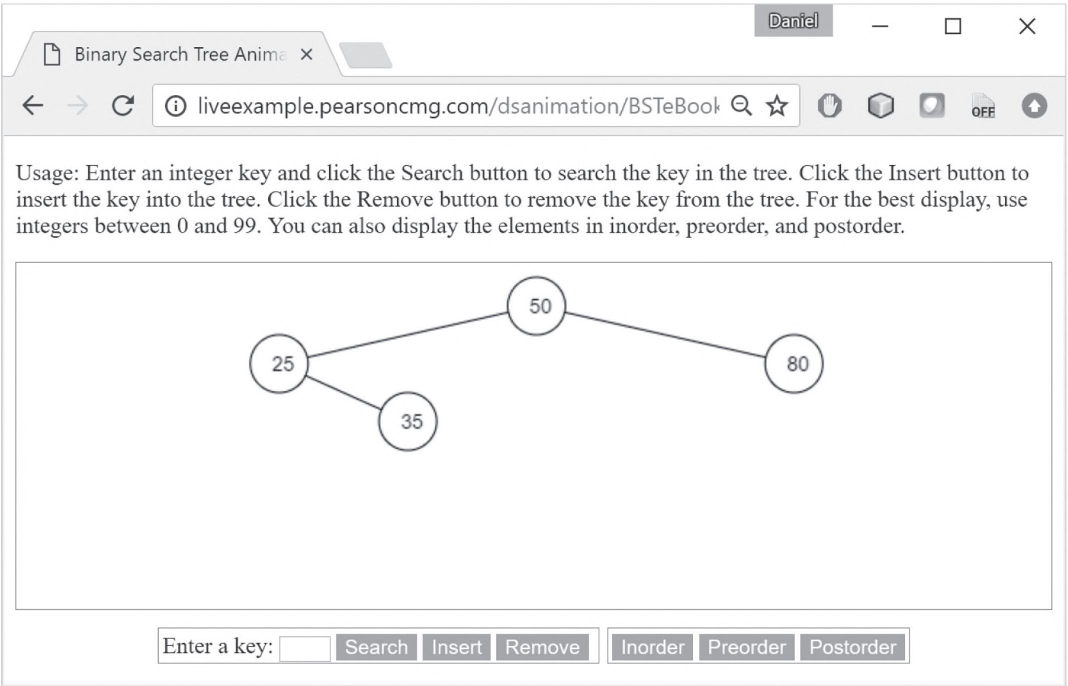
binary search tree

A special type of binary tree called a *binary search tree* (BST) is often useful. A BST (with no duplicate elements) has the property that for every node in the tree, the value of its left child is less than the value of the node, and the value of its right child is greater than the value of the node. The binary trees in Figures 25.1b–c are all BSTs.

BST animation on Companion Website

Pedagogical Note

For an interactive GUI demo to see how a BST works, go to liveexample.pearsoncmg.com/dsanimation/BSTeBook.html, as shown in Figure 25.2.


Daniel
Binary Search Tree Anima
liveexample.pearsoncmg.com/dsanimation/BSTeBook

Usage: Enter an integer key and click the Search button to search the key in the tree. Click the Insert button to insert the key into the tree. Click the Remove button to remove the key from the tree. For the best display, use integers between 0 and 99. You can also display the elements in inorder, preorder, and postorder.

50
25
80
35
Enter a key:
Search
Insert
Remove
Inorder
Preorder
Postorder

FIGURE 25.2 The animation tool enables you to insert, delete, and search elements. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

25.3 Representing Binary Search Trees

A binary search tree can be implemented using a linked structure.

Key Point

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named *left* and *right* that reference the left child and right child, respectively, as shown in Figure 25.3.

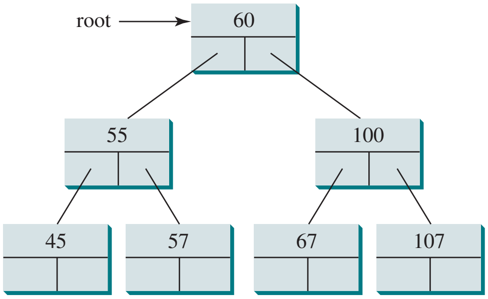

root
60
55
100
45
57
67
107

FIGURE 25.3 A binary tree can be represented using a set of linked nodes.

A node can be defined as a class, as follows:

```
class TreeNode<E> {
  protected E element;
  protected TreeNode<E> left;
  protected TreeNode<E> right;

  public TreeNode(E e) {
    element = e;
  }
}
```

We use the variable `root` to refer to the root node of the tree. If the tree is empty, `root` is `null`. The following code creates the first three nodes of the tree in Figure 25.3:

```
// Create the root node
TreeNode<Integer> root = new TreeNode<>(60);

// Create the left child node
root.left = new TreeNode<>(55);

// Create the right child node
root.right = new TreeNode<>(100);
```

25.4 Searching for an Element

Key Point

BST enables an efficient search that resembles a binary search.

To search for an element in the BST, you start from the root and scan down from it until a match is found or you arrive at an empty subtree. The algorithm is described in Listing 25.1. Let `current` point to the root (line 2). Repeat the following steps until `current` is `null` (line 4) or the element matches `current.element` (line 12):

- If `e` is less than `current.element`, assign `current.left` to `current` (line 6).
- If `e` is greater than `current.element`, assign `current.right` to `current` (line 9).
- If `e` is equal to `current.element`, return `true` (line 12).

If `current` is `null`, the subtree is empty and the element is not in the tree (line 14).

LISTING 25.1 Searching for an Element e in a BST

start from root

left subtree

right subtree

found

not found

```
 1  public boolean search(E e) {
 2    TreeNode<E> current = root; // Start from the root
 3
 4    while (current != null)
 5      if (e < current.element) {
 6        current = current.left; // Go left
 7      }
 8      else if (e > current.element) {
 9        current = current.right; // Go right
10      }
11      else // Element e matches current.element
12        return true; // Element e is found
13
14    return false; // Element e is not in the tree
15  }
```

Check Point

25.4.1 Implement the `search(element)` method using recursion.

25.5 Inserting an Element into a BST

Key Point

The new element is inserted at a leaf node.

To insert an element into a BST, you need to locate where to insert it in the tree. The key idea is to locate the parent for the new node. Listing 25.2 gives the algorithm.

LISTING 25.2 Inserting an Element into a BST

create a new node

```
boolean insert(E e) {
  if (tree is empty)
    // Create the node for e as the root;
  else {
```

```
 5       // Locate the parent node
 6       parent = current = root;
 7       while (current != null)
 8         if (e < the value in current.element) {
 9           parent = current; // Keep the parent
10           current = current.left; // Go left
11         }
12         else if (e > the value in current.element) {
13           parent = current; // Keep the parent
14           current = current.right; // Go right
15         }
16         else
17           return false; // Duplicate node not inserted
18
19       // Create a new node for e and attach it to parent
20
21       return true; // Element inserted
22     }
23   }
```

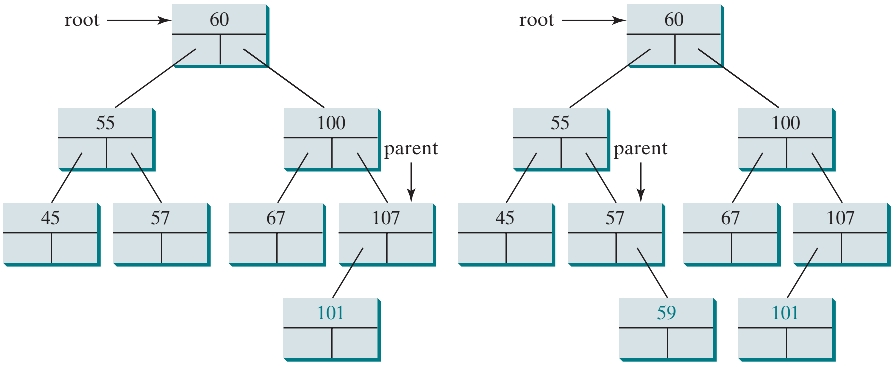
locate parent

left child

right child

If the tree is empty, create a root node with the new element (lines 2 and 3). Otherwise, locate the parent node for the new element node (lines 6–17). Create a new node for the element and link this node to its parent node. If the new element is less than the parent element, the node for the new element will be the left child of the parent. If the new element is greater than the parent element, the node for the new element will be the right child of the parent.

For example, to insert `101` into the tree in Figure 25.3, after the `while` loop finishes in the algorithm, `parent` points to the node for `107`, as shown in Figure 25.4a. The new node for `101` becomes the left child of the parent. To insert `59` into the tree, after the `while` loop finishes in the algorithm, the parent points to the node for `57`, as shown in Figure 25.4b. The new node for `59` becomes the right child of the parent.



(a) Inserting 101 (b) Inserting 59

FIGURE 25.4 Two new elements are inserted into the tree.

25.5.1 Show the result of inserting 44 into Figure 25.4b.

25.5.2 What is the time complexity of inserting an element into a BST?

Check Point

25.6 Tree Traversal

Inorder, preorder, postorder, depth-first, and breadth-first are common ways to traverse the elements in a binary tree.

Key Point

Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree. This section presents *inorder*, *postorder, preorder*, *depth-first,* and *breadth-first* traversals.

tree traversal

inorder traversal

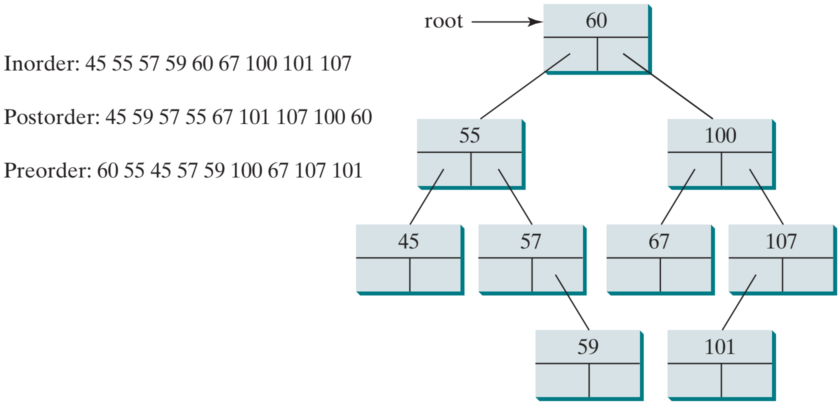
With *inorder traversal*, the left subtree of the current node is visited first recursively, then the current node, and finally the right subtree of the current node recursively. The inorder traversal displays all the nodes in a BST in increasing order, as shown in Figure 25.5.

postorder traversal

With *postorder traversal*, the left subtree of the current node is visited recursively first, then recursively the right subtree of the current node, and finally the current node itself.

preorder traversal

With *preorder traversal*, the current node is visited first, then recursively the left subtree of the current node, and finally the right subtree of the current node recursively.


Inorder: 45 55 57 59 60 67 100 101 107

Postorder: 45 59 57 55 67 101 107 100 60

Preorder: 60 55 45 57 59 100 67 107 101

FIGURE 25.5 A tree traversal visits each node in a tree in a certain order.

Note

reconstruct a tree

You can reconstruct a binary search tree by inserting the elements in their preorder. The reconstructed tree preserves the parent and child relationship for the nodes in the original binary search tree.

depth-first traversal

Depth-first traversal is to visit the root then recursively visit its left subtree and right subtree in an arbitrary order. The preorder traversal can be viewed as a special case of depth-first traversal, which recursively visits its left subtree then its right subtree.

breadth-first traversal

With *breadth-first traversal*, the nodes are visited level by level. First the root is visited, then all the children of the root from left to right, then the grandchildren of the root from left to right, and so on.

For example, in the tree in Figure 25.5, the inorder is

```
45 55 57 59 60 67 100 101 107
```

The postorder is

```
45 59 57 55 67 101 107 100 60
```

The preorder is

```
60 55 45 57 59 100 67 107 101
```

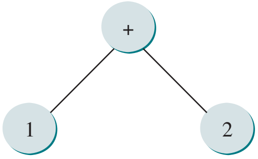
The depth-first is

```
60 55 45 57 59 100 67 107 101
```

The breadth-first traversal is

```
60 55 100 45 57 67 107 59 101
```

You can use the following simple tree to help remember inorder, postorder, and preorder.



The inorder is `1 + 2`, the postorder is `1 2 +`, and the preorder is `+ 1 2`.

Check Point

25.6.1 Show the inorder, preorder, and postorder of traversing the elements in the binary tree shown in Figure 25.1c?

25.6.2 If a set of elements is inserted into a BST in two different orders, will the two corresponding BSTs look the same? Will the inorder traversal be the same? Will the postorder traversal be the same? Will the preorder traversal be the same?

25.7 The BST Class

The **BST** *class defines a data structure for storing and manipulating data in a binary search tree.*

Key Point

Following the design pattern for Java Collections Framework and utilizing the default methods in Java 8, we use an interface named `Tree` to define all common operations for trees and define `Tree` to be a subtype of `Collection` so we can use common operations in `Collection` for trees, as shown in Figure 25.6. A concrete `BST` class can be defined to implement `Tree`, as shown in Figure 25.7.

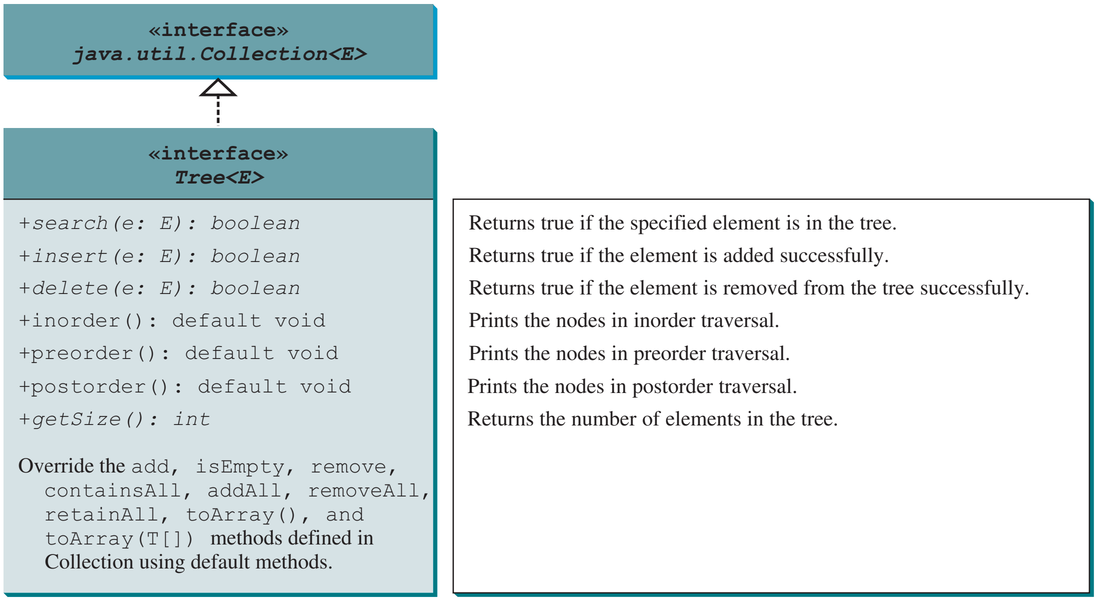

«interface»
java.util.Collection<E>
«interface»
Tree<E>
+search(e: E): boolean
+insert(e: E): boolean
+delete(e: E): boolean
+inorder(): default void
+preorder(): default void
+postorder(): default void
+getSize(): int
Override the add, isEmpty, remove, containsAll, addAll, removeAll, retainAll, toArray(), and toArray(T[]) methods defined in Collection using default methods.
Returns true if the specified element is in the tree.
Returns true if the element is added successfully.
Returns true if the element is removed from the tree successfully.
Prints the nodes in inorder traversal.
Prints the nodes in preorder traversal.
Prints the nodes in postorder traversal.
Returns the number of elements in the tree.

FIGURE 25.6 The `Tree` interface defines common operations for trees, and partially implements `Collection`.

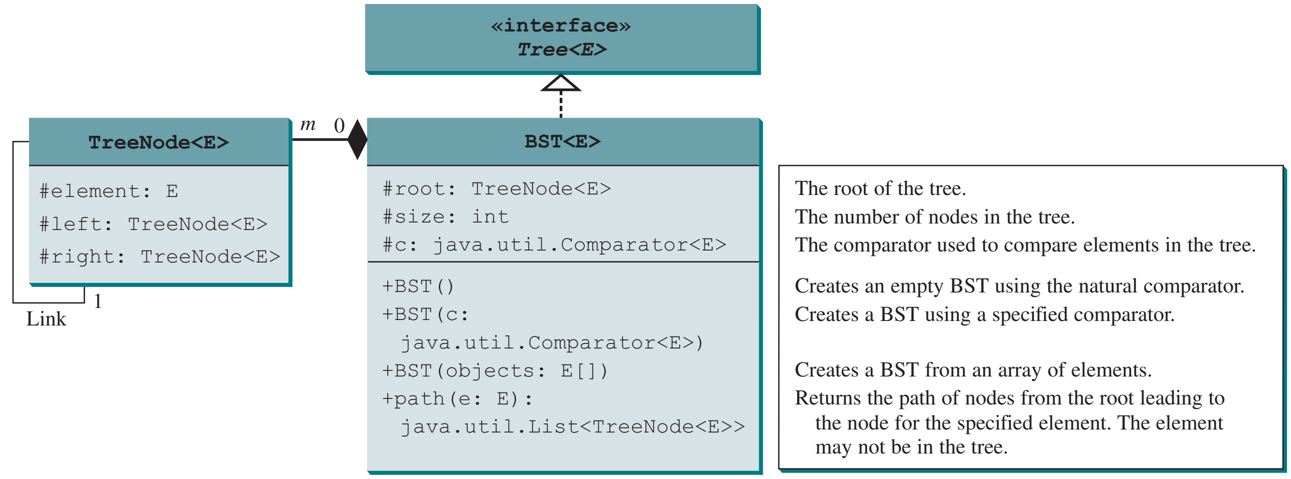

«interface»
Tree<E>
TreeNode<E>
#element: E
#left: TreeNode<E>
#right: TreeNode<E>
m
0
1
Link
BST<E>
#root: TreeNode<E>
#size: int
#c: java.util.Comparator<E>
+BST()
+BST(c: java.util.Comparator<E>)
+BST(objects: E[])
+path(e: E): java.util.List<TreeNode<E>>
The root of the tree.
The number of nodes in the tree.
The comparator used to compare elements in the tree.
Creates an empty BST using the natural comparator.
Creates a BST using a specified comparator.
Creates a BST from an array of elements.
Returns the path of nodes from the root leading to the node for the specified element. The element may not be in the tree.

FIGURE 25.7 The **`BST`** class defines a concrete BST.

Listing 25.3 gives the implementation for **`Tree`**. It provides default implementations for the **`add`**, **`isEmpty`**, **`remove`**, **`containsAll`**, **`addAll`**, **`removeAll`**, **`retainAll`**, **`toArray()`**, and **`toArray(T[])`** methods inherited from the **`Collection`** interface as well as the **`inorder()`**, **`preorder()`**, and **`postorder()`** defined in the **`Tree`** interface.

LISTING 25.3 `Tree.java`

interface
search
insert
delete
getSize
inorder
postorder
preorder
default isEmpty

```java
import java.util.Collection;

public interface Tree<E> extends Collection<E> {
  /** Return true if the element is in the tree */
  public boolean search(E e);

  /** Insert element e into the binary tree
   * Return true if the element is inserted successfully */
  public boolean insert(E e);

  /** Delete the specified element from the tree
   * Return true if the element is deleted successfully */
  public boolean delete(E e);

  /** Get the number of elements in the tree */
  public int getSize();

  /** Inorder traversal from the root*/
  public default void inorder() {
  }

  /** Postorder traversal from the root */
  public default void postorder() {
  }

  /** Preorder traversal from the root */
  public default void preorder() {
  }

  @Override /** Return true if the tree is empty */
  public default boolean isEmpty() {
    return size() == 0;
  }

  @Override
  public default boolean contains(Object e) {
    return search((E)e);
  }

  @Override
  public default boolean add(E e) {
    return insert(e);
  }

  @Override
  public default boolean remove(Object e) {
    return delete((E)e);
  }

  @Override
  public default int size() {
    return getSize();
  }

  @Override
  public default boolean containsAll(Collection<?> c) {
    // Left as an exercise
    return false;
  }

  @Override
  public default boolean addAll(Collection<? extends E> c) {
    // Left as an exercise
    return false;
  }

  @Override
  public default boolean removeAll(Collection<?> c) {
    // Left as an exercise
    return false;
  }

  @Override
  public default boolean retainAll(Collection<?> c) {
    // Left as an exercise
    return false;
  }

  @Override
  public default Object[] toArray() {
    // Left as an exercise
    return null;
  }

  @Override
  public default <T> T[] toArray(T[] array) {
    // Left as an exercise
    return null;
  }
}
```

default `contains`

default `add`

default `remove`

default `size`

default `containsAll`

default `addAll`

default `removeAll`

default `retainAll`

default `toArray()`

default toArray(T[])

Listing 25.4 gives the implementations for the **BST** class.

LISTING 25.4 BST.java

BST class
root
size
comparator
no-arg constructor
natural order comparator
constructor
constructor
search
compare objects
insert
new root
compare objects

```
public class BST<E> implements Tree<E> {
  protected TreeNode<E> root;
  protected int size = 0;
  protected java.util.Comparator<E> c;

  /** Create a default BST with a natural order comparator */
  public BST() {
    this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
  }

  /** Create a BST with a specified comparator */
  public BST(java.util.Comparator<E> c) {
    this.c = c;
  }

  /** Create a binary tree from an array of objects */
  public BST(E[] objects) {
    this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
    for (int i = 0; i < objects.length; i++)
      add(objects[i]);
  }

  @Override /** Return true if the element is in the tree */
  public boolean search(E e) {
    TreeNode<E> current = root; // Start from the root

    while (current != null) {
      if (c.compare(e, current.element) < 0) {
        current = current.left;
      }
      else if (c.compare(e, current.element) > 0) {
        current = current.right;
      }
      else // element matches current.element
        return true; // Element is found
    }

    return false;
  }

  @Override /** Insert element e into the binary tree
   * Return true if the element is inserted successfully */
  public boolean insert(E e) {
    if (root == null)
      root = createNewNode(e); // Create a new root
    else {
      // Locate the parent node
      TreeNode<E> parent = null;
      TreeNode<E> current = root;
      while (current != null)
        if (c.compare(e, current.element) < 0) {
          parent = current;
          current = current.left;
        }
        else if (c.compare(e, current.element) > 0) {
          parent = current;
          current = current.right;
        }
        else
```

```
          return false; // Duplicate node not inserted

      // Create the new node and attach it to the parent node
      if (c.compare(e, parent.element) < 0)
        parent.left = createNewNode(e);
      else
        parent.right = createNewNode(e);
    }

    size++;
    return true; // Element inserted successfully
  }

  protected TreeNode<E> createNewNode(E e) {
    return new TreeNode<>(e);
  }

  @Override /** Inorder traversal from the root */
  public void inorder() {
    inorder(root);
  }

  /** Inorder traversal from a subtree */
  protected void inorder(TreeNode<E> root) {
    if (root == null) return;
    inorder(root.left);
    System.out.print(root.element + " ");
    inorder(root.right);
  }

  @Override /** Postorder traversal from the root */
  public void postorder() {
    postorder(root);
  }

  /** Postorder traversal from a subtree */
  protected void postorder(TreeNode<E> root) {
    if (root == null) return;
    postorder(root.left);
    postorder(root.right);
    System.out.print(root.element + " ");
  }

  @Override /** Preorder traversal from the root */
  public void preorder() {
    preorder(root);
  }

  /** Preorder traversal from a subtree */
  protected void preorder(TreeNode<E> root) {
    if (root == null) return;
    System.out.print(root.element + " ");
    preorder(root.left);
    preorder(root.right);
  }

  /** This inner class is static, because it does not access
      any instance members defined in its outer class */
  public static class TreeNode<E> {
    protected E element;
    protected TreeNode<E> left;
```

link to parent

increase size

create new node

`inorder`

recursive helper method

`postorder`

recursive helper method

`preorder`

recursive helper method

inner class

```
    protected TreeNode<E> right;

    public TreeNode(E e) {
      element = e;
    }
  }

  @Override /** Get the number of nodes in the tree */
```

getSize

```
  public int getSize() {
    return size;
  }

  /** Returns the root of the tree */
```

getRoot

```
  public TreeNode<E> getRoot() {
    return root;
  }

  /** Returns a path from the root leading to the specified element */
```

path

```
  public java.util.ArrayList<TreeNode<E>> path(E e) {
    java.util.ArrayList<TreeNode<E>> list =
      new java.util.ArrayList<>();
    TreeNode<E> current = root; // Start from the root

    while (current != null) {
      list.add(current); // Add the node to the list
      if (c.compare(e, current.element) < 0) {
        current = current.left;
      }
      else if (c.compare(e, current.element) > 0) {
        current = current.right;
      }
      else
        break;
    }

    return list; // Return an array list of nodes
  }

  @Override /** Delete an element from the binary tree.
   * Return true if the element is deleted successfully
   * Return false if the element is not in the tree */
```

delete

```
  public boolean delete(E e) {
    // Locate the node to be deleted and also locate its parent node
```

locate parent

```
    TreeNode<E> parent = null;
```

locate current

```
    TreeNode<E> current = root;
    while (current != null) {
      if (c.compare(e, current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (c.compare(e, current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
```

current found

```
        break; // Element is in the tree pointed at by current
    }

```

not found

```
    if (current == null)
```

```
      return false; // Element is not in the tree

    // Case 1: current has no left child
    if (current.left == null) {
     // Connect the parent with the right child of the current node
      if (parent == null) {
        root = current.right;
      }
      else {
        if (c.compare(e, parent.element) < 0)
          parent.left = current.right;
        else
          parent.right = current.right;
      }
    }
    else {
      // Case 2: The current node has a left child
      // Locate the rightmost node in the left subtree of
      // the current node and also its parent
      TreeNode<E> parentOfRightMost = current;
      TreeNode<E> rightMost = current.left;

      while (rightMost.right != null) {
        parentOfRightMost = rightMost;
        rightMost = rightMost.right; // Keep going to the right
      }

      // Replace the element in current by the element in rightMost
      current.element = rightMost.element;

      // Eliminate rightmost node
      if (parentOfRightMost.right == rightMost)
        parentOfRightMost.right = rightMost.left;
      else
        // Special case: parentOfRightMost == current
        parentOfRightMost.left = rightMost.left;
    }

    size--;
    return true; // Element deleted successfully
  }

  @Override /** Obtain an iterator. Use inorder. */
  public java.util.Iterator<E> iterator() {
    return new InorderIterator();
  }

  // Inner class InorderIterator
  private class InorderIterator implements java.util.Iterator<E> {
    // Store the elements in a list
    private java.util.ArrayList<E> list =
      new java.util.ArrayList<>();
   private int current = 0; // Point to the current element in list

    public InorderIterator() {
     inorder(); // Traverse binary tree and store elements in list
    }

    /** Inorder traversal from the root*/
    private void inorder() {
      inorder(root);
```

Case 1

reconnect parent

reconnect parent

Case 2

locate `parentOfRightMost`

locate `rightMost`

replace current

reconnect `parentOfRightMost`

reduce size

successful deletion

iterator

iterator class

internal list

current position

obtain inorder list

```
241     }
242 
243     /** Inorder traversal from a subtree */
244     private void inorder(TreeNode<E> root) {
245       if (root == null) return;
246       inorder(root.left);
247       list.add(root.element);
248       inorder(root.right);
249     }
250 
251     @Override /** More elements for traversing? */
252     public boolean hasNext() {
253       if (current < list.size())
254         return true;
255 
256       return false;
257     }
258 
259     @Override /** Get the current element and move to the next */
260     public E next() {
261       return list.get(current++);
262     }
263 
264     @Override // Remove the element returned by the last next()
265     public void remove() {
266        if (current == 0) // next() has not been called yet
267         throw new IllegalStateException();
268 
269        delete(list.get(--current));
270       list.clear(); // Clear the list
271       inorder(); // Rebuild the list
272     }
273   }
274 
275   @Override /** Remove all elements from the tree */
276   public void clear() {
277     root = null;
278     size = 0;
279   }
280 }
```

`hasNext` in iterator?

get next element

remove the current

`clear` list

refresh list

tree clear method

The `insert(E e)` method (lines 43–71) creates a node for element `e` and inserts it into the tree. If the tree is empty, the node becomes the root. Otherwise, the method finds an appropriate parent for the node to maintain the order of the tree. If the element is already in the tree, the method returns `false`; otherwise it returns `true`.

The `inorder()` method (lines 78–88) invokes `inorder(root)` to traverse the entire tree. The method `inorder(TreeNode root)` traverses the tree with the specified root. This is a recursive method. It recursively traverses the left subtree, then the root, and finally the right subtree. The traversal ends when the tree is empty.

The `postorder()` method (lines 91–101) and the `preorder()` method (lines 104–114) are implemented similarly using recursion.

The `path(E e)` method (lines 139–157) returns a path of the nodes as an array list. The path starts from the root leading to the element. The element may not be in the tree. For example, in Figure 25.4a, `path(45)` contains the nodes for elements `60`, `55`, and `45`, and `path(58)` contains the nodes for elements `60`, `55`, and `57`.

The implementation of `delete()` and `iterator()` (lines 162–273) will be discussed in Sections 25.8 and 25.10.

Design Pattern Note

The design for the `createNewNode()` method applies the factory method pattern, which creates an object returned from the method rather than directly using the constructor in the code to create the object. Suppose the factory method returns an object of the type `A`. This design enables you to override the method to create an object of the subtype of `A`.The `createNewNode()` method in the `BST` class returns a `TreeNode` object. Later in later chapters, we will override this method to return an object of the subtype of `TreeNode`.

Listing 25.5 gives an example that creates a binary search tree using `BST` (line 4). The program adds strings into the tree (lines 5–11), traverses the tree in inorder, postorder, and preorder (lines 14–20), searches for an element (line 24), and obtains a path from the node containing `Peter` to the root (lines 28–31).

LISTING 25.5 `TestBST.java`

```
 1  public class TestBST {
 2    public static void main(String[] args) {
 3      // Create a BST
 4      BST<String> tree = new BST<>();
 5      tree.insert("George");
 6      tree.insert("Michael");
 7      tree.insert("Tom");
 8      tree.insert("Adam");
 9      tree.insert("Jones");
10      tree.insert("Peter");
11      tree.insert("Daniel");
12
13      // Traverse tree
14      System.out.print("Inorder (sorted): ");
15      tree.inorder();
16      System.out.print("\nPostorder: ");
17      tree.postorder();
18      System.out.print("\nPreorder: ");
19      tree.preorder();
20      System.out.print("\nThe number of nodes is " + tree.getSize());
21
22      // Search for an element
23      System.out.print("\nIs Peter in the tree? " +
24        tree.search("Peter"));
25
26      // Get a path from the root to Peter
27      System.out.print("\nA path from the root to Peter is: ");
28      java.util.ArrayList<BST.TreeNode<String>> path
29        = tree.path("Peter");
30      for (int i = 0; path != null && i < path.size(); i++)
31        System.out.print(path.get(i).element + " ");
32
33      Integer[] numbers = {2, 4, 3, 1, 8, 5, 6, 7};
34      BST<Integer> intTree = new BST<>(numbers);
35      System.out.print("\nInorder (sorted): ");
36      intTree.inorder();
37    }
38  }
```

create tree

`insert`

`inorder`

`postorder`

`preorder`

`getSize`

`search`

```
Inorder (sorted): Adam Daniel George Jones Michael Peter Tom
Postorder: Daniel Adam Jones Peter Tom Michael George
Preorder: George Adam Daniel Michael Jones Tom Peter
The number of nodes is 7
Is Peter in the tree? true
A path from the root to Peter is: George Michael Tom Peter
Inorder (sorted): 1 2 3 4 5 6 7 8
```

The program checks `path != null` in line 30 to ensure that the path is not `null` before invoking `path.get(i)`. This is an example of defensive programming to avoid potential runtime errors.

The program creates another tree for storing `int` values (line 34). After all the elements are inserted in the trees, the trees should appear as shown in Figure 25.8.

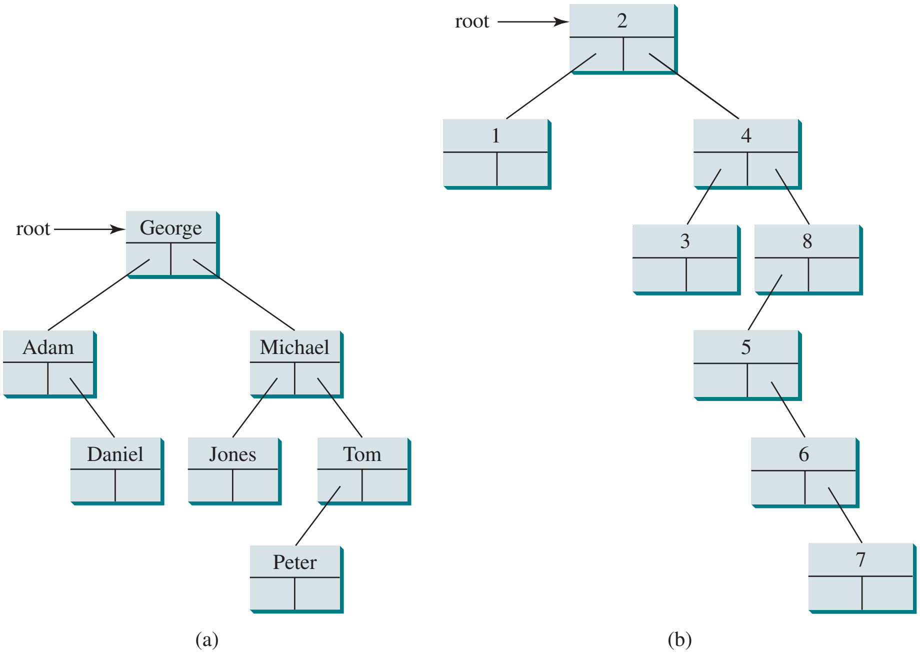


FIGURE 25.8 The BSTs in Listing 25.5 are pictured here after they are created.

If the elements are inserted in a different order (e.g., Daniel, Adam, Jones, Peter, Tom, Michael, and George), the tree will look different. However, the inorder traversal prints elements in the same order as long as the set of elements is the same. The inorder traversal displays a sorted list.

Check Point

25.7.1 Write the code that creates a `BST` for integers, inserts numbers `9`, `3`, `12`, `30` to the tree, and deletes `9`, display the tree in order, preorder, and postorder, searches for `4`, and displays the number of elements in the tree.

25.7.2 Add a new method named `getSmallest()` in the `BST` class that returns the smallest element in the tree.

25.7.3 Add a new method named `getLargest()` in the `BST` class that returns the largest element in the tree.

25.8 Deleting Elements from a BST

Key Point

To delete an element from a BST, first locate it in the tree then consider two cases—whether or not the node has a left child—before deleting the element and reconnecting the tree.

The `insert(element)` method is presented in Section 25.5. Often, you need to delete an element from a binary search tree. Doing so is far more complex than adding an element into a binary search tree.

To delete an element from a binary search tree, you need to first locate the node that contains the element and also its parent node. Let `current` point to the node that contains the element in the binary search tree and `parent` point to the parent of the `current` node. The `current` node may be a left child or a right child of the `parent` node. There are two cases to consider.

locating element

Case 1: The current node does not have a left child, as shown in Figure 25.9a. In this case, simply connect the parent with the right child of the current node, as shown in Figure 25.9b.

For example, to delete node `10` in Figure 25.10a, you would connect the parent of node `10` with the right child of node `10`, as shown in Figure 25.10b.

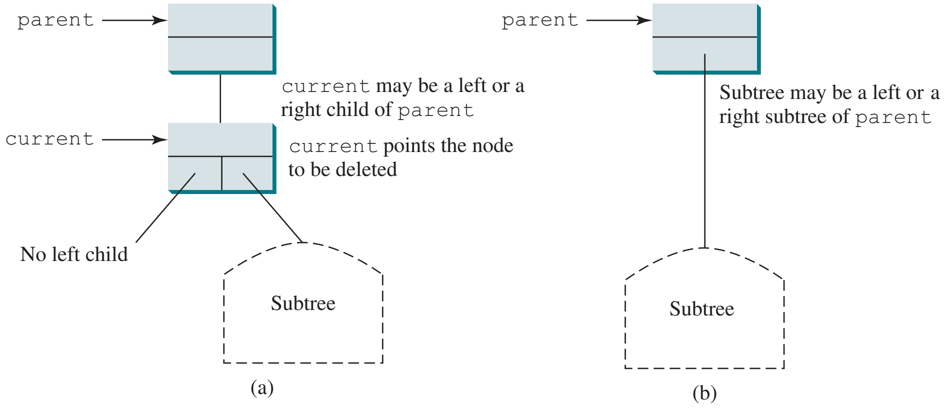

parent
current may be a left or a right child of parent
current
current points the node to be deleted
No left child
Subtree
(a)
parent
Subtree may be a left or a right subtree of parent
Subtree
(b)

FIGURE 25.9 Case 1: The current node has no left child.

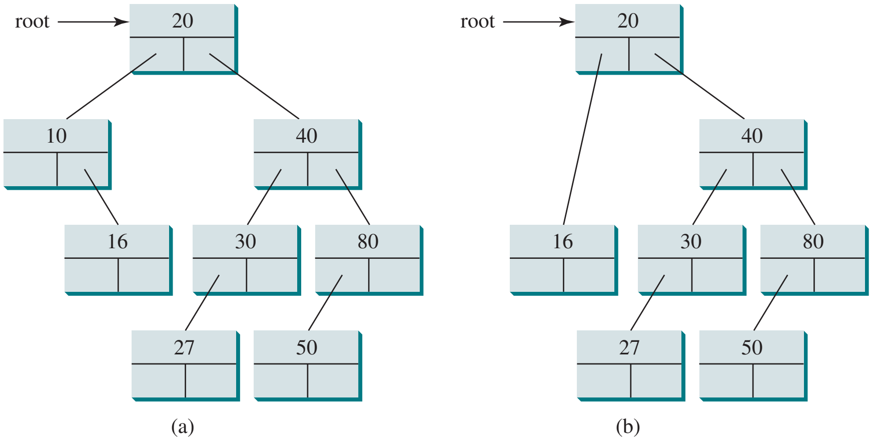

root
20
10
40
16
30
80
27
50
(a)
root
20
40
16
30
80
27
50
(b)

FIGURE 25.10 Case 1: Deleting node `10` from (a) results in (b).

Note

If the current node is a leaf, it falls into Case 1. For example, to delete element `16` in Figure 25.10a, connect its right child (in this case, it is `null`) to the parent of node `16`.

delete a leaf

Case 2: The `current` node has a left child. Let `rightMost` point to the node that contains the largest element in the left subtree of the `current` node and `parentOfRightMost` point to the parent node of the `rightMost` node, as shown in Figure 25.11a. Note the `rightMost` node cannot have a right child but may have a left child. Replace the element value in the `current` node with the one in the `rightMost` node, connect the `parentOfRightMost` node with the left child of the `rightMost` node, and delete the `rightMost` node, as shown in Figure 25.11b.

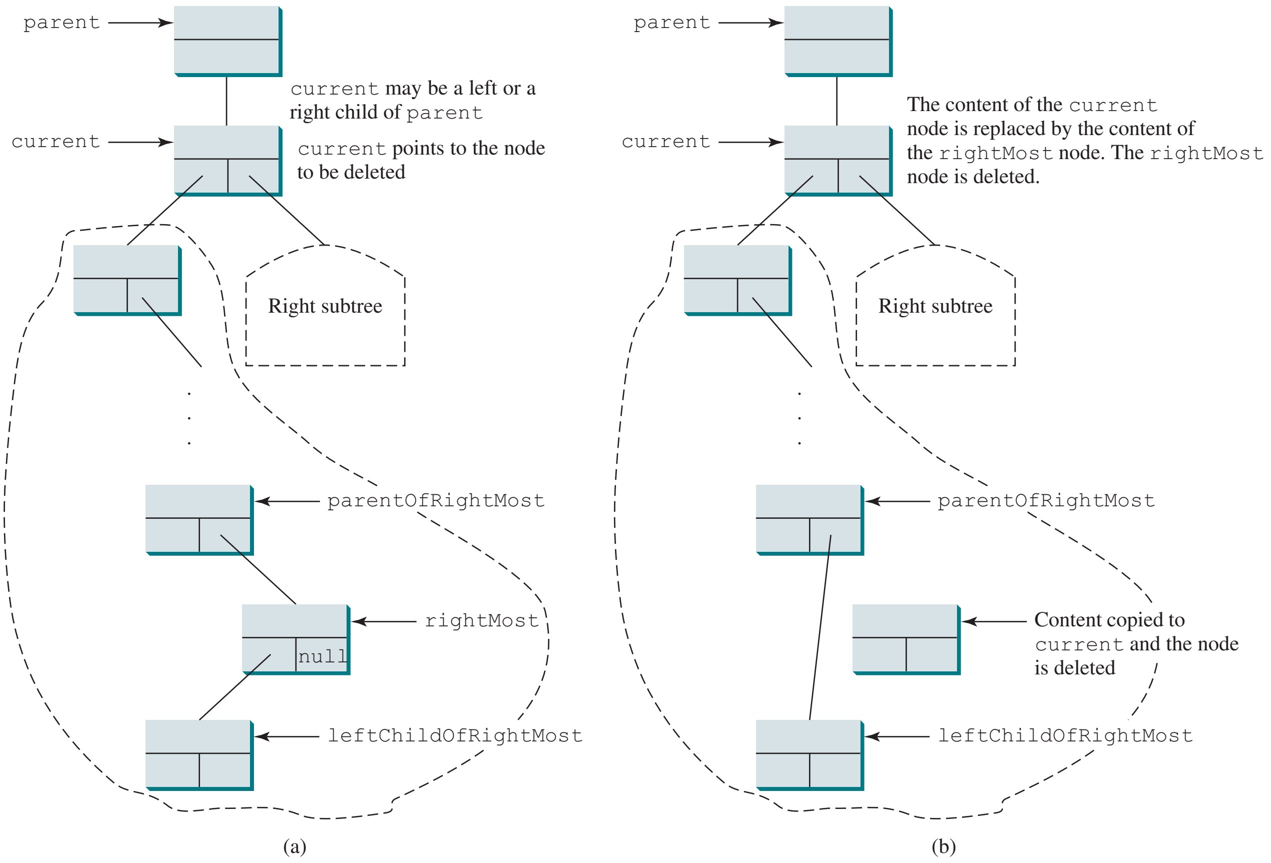

parent
current
`current` may be a left or a right child of `parent`
`current` points to the node to be deleted
Right subtree
parentOfRightMost
rightMost
null
leftChildOfRightMost
(a)
parent
current
The content of the `current` node is replaced by the content of the `rightMost` node. The `rightMost` node is deleted.
Right subtree
parentOfRightMost
Content copied to `current` and the node is deleted
leftChildOfRightMost
(b)

FIGURE 25.11 Case 2: The current node has a left child.

For example, consider deleting node **`20`** in Figure 25.12a. The **`rightMost`** node has the element value **`16`**. Replace the element value **`20`** with **`16`** in the **`current`** node and make node **`10`** the parent for node **`14`**, as shown in Figure 25.12b.

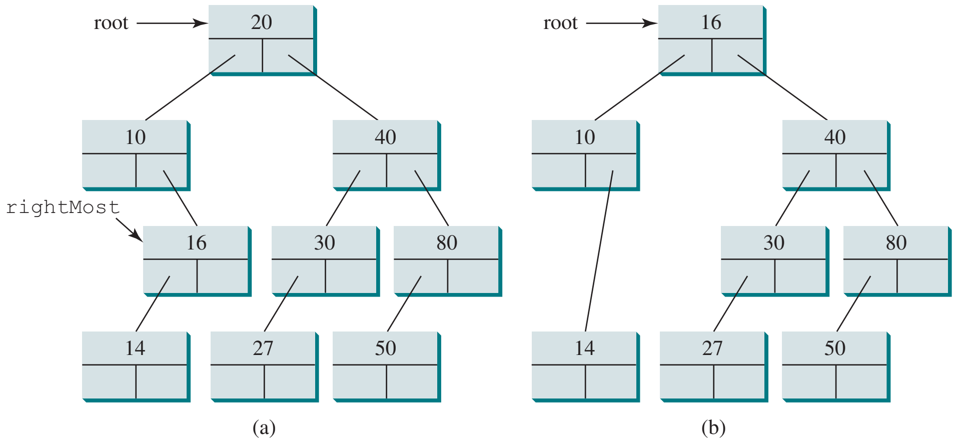

root
20
10
40
rightMost
16
30
80
14
27
50
(a)
root
16
10
40
30
80
14
27
50
(b)

FIGURE 25.12 Case 2: Deleting node **`20`** from (a) results in (b).

Note

If the left child of **`current`** does not have a right child, **`current.left`** points to the largest element in the left subtree of **`current`**. In this case, **`rightMost`** is **`current.left`** and **`parentOfRightMost`** is **`current`**. You have to take care of this special case to reconnect the left child of **`rightMost`** with **`parentOfRightMost`**.

special case

The algorithm for deleting an element from a binary search tree can be described in Listing 25.6.

`delete` method

Listing 25.6 Deleting an Element from a BST

```
boolean delete(E e) {
  Locate element e in the tree;
  if element e is not found
    return false;

  Let current be the node that contains e and parent be
    the parent of current;

  if (current has no left child)  // Case 1
    Connect the right child of current with parent;
      Now current is not referenced, so it is eliminated;
  else // Case 2
    Locate the rightmost node in the left subtree of current.
    Copy the element value in the rightmost node to current.
    Connect the parent of the rightmost node to the left child
      of rightmost node;

  return true; // Element deleted
}
```

not in the tree

locate current

locate parent

Case 1

Case 2

The complete implementation of the **`delete`** method is given in lines 162–220 in Listing 25.4. The method locates the node (named **`current`**) to be deleted and also locates its parent (named **`parent`**) in lines 164–177. If **`current`** is **`null`**, the element is not in the tree. Therefore, the method returns **`false`** (line 180). Please note that if **`current`** is **`root`**, **`parent`** is **`null`**. If the tree is empty, both **`current`** and **`parent`** are **`null`**.

Case 1 of the algorithm is covered in lines 183–194. In this case, the **`current`** node has no left child (i.e., **`current.left == null`**). If **`parent`** is **`null`**, assign **`current.right`** to **`root`** (lines 185–187). Otherwise, assign **`current.right`** to either **`parent.left`** or **`parent.right`**, depending on whether **`current`** is a left or a right child of **`parent`** (189–192).

Case 2 of the algorithm is covered in lines 195–216. In this case, **`current`** has a left child. The algorithm locates the rightmost node (named **`rightMost`**) in the left subtree of the current node and also its parent (named **`parentOfRightMost`**) (lines 199–205). Replace the element in **`current`** by the element in **`rightMost`** (line 208); assign **`rightMost.left`** to either **`parentOfRightMost.right`** or **`parentOfRightMost.left`** (lines 211–215), depending on whether **`rightMost`** is a right or a left child of **`parentOfRightMost`**.

Listing 25.7 gives a test program that deletes the elements from the binary search tree.

Listing 25.7 TestBSTDelete.java

```java
public class TestBSTDelete {
  public static void main(String[] args) {
    BST<String> tree = new BST<>();
    tree.insert("George");
    tree.insert("Michael");
    tree.insert("Tom");
    tree.insert("Adam");
    tree.insert("Jones");
    tree.insert("Peter");
    tree.insert("Daniel");
    printTree(tree);

    System.out.println("\nAfter delete George:");
```

delete an element

delete an element

delete an element

```
    tree.delete("George");
    printTree(tree);

    System.out.println("\nAfter delete Adam:");
    tree.delete("Adam");
    printTree(tree);

    System.out.println("\nAfter delete Michael:");
    tree.delete("Michael");
    printTree(tree);
  }

  public static void printTree(BST tree) {
    // Traverse tree
    System.out.print("Inorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```

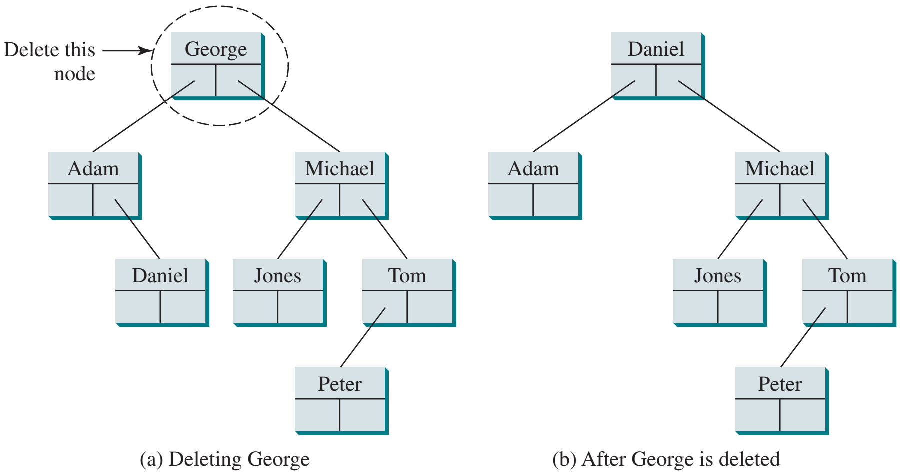
```
Inorder (sorted): Adam Daniel George Jones Michael Peter Tom
Postorder: Daniel Adam Jones Peter Tom Michael George
Preorder: George Adam Daniel Michael Jones Tom Peter
The number of nodes is 7

After delete George:
Inorder (sorted): Adam Daniel Jones Michael Peter Tom
Postorder: Adam Jones Peter Tom Michael Daniel
Preorder: Daniel Adam Michael Jones Tom Peter
The number of nodes is 6

After delete Adam:
Inorder (sorted): Daniel Jones Michael Peter Tom
Postorder: Jones Peter Tom Michael Daniel
Preorder: Daniel Michael Jones Tom Peter
The number of nodes is 5

After delete Michael:
Inorder (sorted): Daniel Jones Peter Tom
Postorder: Peter Tom Jones Daniel
Preorder: Daniel Jones Tom Peter
The number of nodes is 4
```

Figures 25.12–25.14 show how the tree evolves as the elements are deleted.

BST time complexity

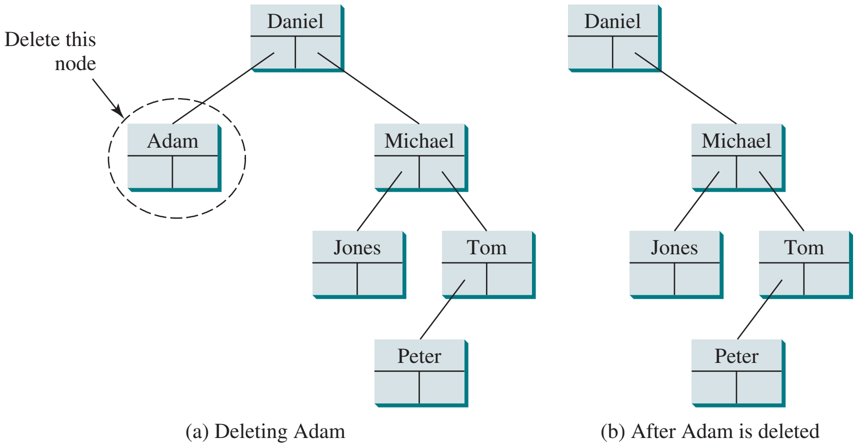
Note

It is obvious that the time complexity for the inorder, preorder, and postorder is $O(n)$, since each node is traversed only once. The time complexity for search, insertion, and deletion is the height of the tree. In the worst case, the height of the tree is $O(n)$. On average, the height of the tree is $O(\log n)$. So, the average time for search, insertion, deletion in a BST is $O(\log n)$.


Delete this node
George
Adam
Michael
Daniel
Jones
Tom
Peter
Daniel
Adam
Michael
Jones
Tom
Peter

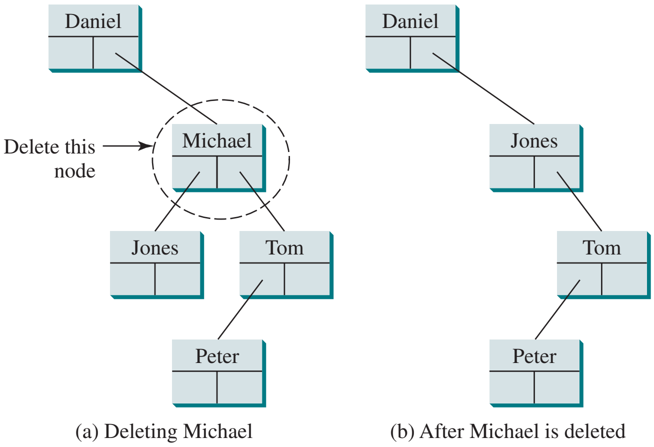
(a) Deleting George (b) After George is deleted

FIGURE 25.13 Deleting George falls into Case 2.


Delete this node
Daniel
Adam
Michael
Jones
Tom
Peter
Daniel
Michael
Jones
Tom
Peter

(a) Deleting Adam (b) After Adam is deleted

FIGURE 25.14 Deleting Adam falls into Case 1.


Daniel
Delete this node
Michael
Jones
Tom
Peter
Daniel
Jones
Tom
Peter

(a) Deleting Michael (b) After Michael is deleted

FIGURE 25.15 Deleting Michael falls into Case 2.

Check Point

25.8.1 Show the result of deleting **55** from the tree in Figure 25.4b.

25.8.2 Show the result of deleting **60** from the tree in Figure 25.4b.

25.8.3 What is the time complexity of deleting an element from a BST?

25.8.4 Is the algorithm correct if lines 203–207 in Listing 25.4 in Case 2 of the **`delete()`** method are replaced by the following code?

```
parentOfRightMost.right = rightMost.left;
```

25.9 Tree Visualization and MVC

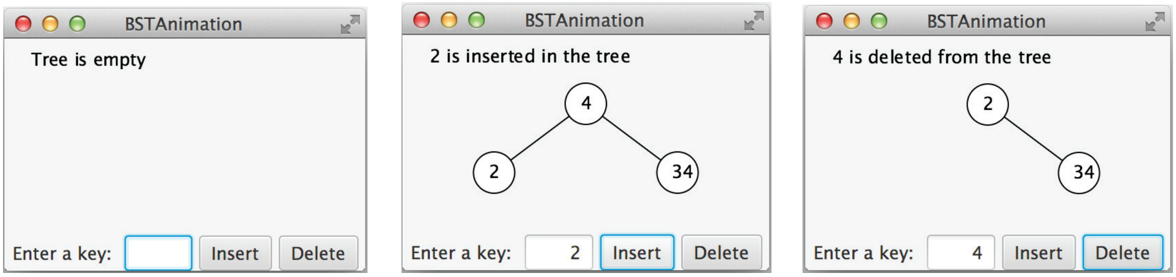
Key Point

You can use recursion to display a binary tree.

Pedagogical Note

One challenge facing the data-structure course is to motivate students. Displaying a binary tree graphically will not only help you understand the working of a binary tree but perhaps also stimulate your interest in programming. This section introduces the techniques to visualize binary trees. You can also apply visualization techniques to other projects.

How do you display a binary tree? It is a recursive structure, so you can display a binary tree using recursion. You can simply display the root, then display the two subtrees recursively. The techniques for displaying the Sierpinski triangle in Listing 18.9 can be applied to displaying a binary tree. For simplicity, we assume that the keys are positive integers less than **100**. Listing 25.8 and Listing 25.9 give the program, and Figure 25.15 shows some sample runs of the program.



Tree is empty

2 is inserted in the tree

4 is deleted from the tree

FIGURE 25.16 A binary tree is displayed graphically. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

LISTING 25.8 BSTAnimation.java

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;

public class BSTAnimation extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    BST<Integer> tree = new BST<>(); // Create a tree

    BorderPane pane = new BorderPane();
    BTView view = new BTView(tree); // Create a View
```

create a tree

view for tree

```
    pane.setCenter(view);

    TextField tfKey = new TextField();
    tfKey.setPrefColumnCount(3);
    tfKey.setAlignment(Pos.BASELINE_RIGHT);
    Button btInsert = new Button("Insert");
    Button btDelete = new Button("Delete");
    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(new Label("Enter a key: "),
      tfKey, btInsert, btDelete);
    hBox.setAlignment(Pos.CENTER);
    pane.setBottom(hBox);

    btInsert.setOnAction(e -> {
      int key = Integer.parseInt(tfKey.getText());
      if (tree.search(key)) { // key is in the tree already
        view.displayTree();
        view.setStatus(key + " is already in the tree");
      }
      else {
        tree.insert(key); // Insert a new key
        view.displayTree();
        view.setStatus(key + " is inserted in the tree");
      }
    });

    btDelete.setOnAction(e -> {
      int key = Integer.parseInt(tfKey.getText());
      if (!tree.search(key)) {  // key is not in the tree
        view.displayTree();
        view.setStatus(key + " is not in the tree");
      }
      else {
        tree.delete(key); // Delete a key
        view.displayTree();
        view.setStatus(key + " is deleted from the tree");
      }
    });

    // Create a scene and place the pane in the stage
    Scene scene = new Scene(pane, 450, 250);
    primaryStage.setTitle("BSTAnimation"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

place tree view

place hBox

handle insertion

insert key

display the tree

handle deletion

delete key

display the tree

main method omitted

LISTING 25.9 `BTView.java`

```
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;

public class BTView extends Pane {
  private BST<Integer> tree = new BST<>();
  private double radius = 15; // Tree node radius
  private double vGap = 50; // Gap between two levels in a tree

```

tree to display

set a tree

clear the display

display tree recursively

connect two nodes

draw left subtree

connect two nodes

draw right subtree

display a node

```
12   BTView(BST<Integer> tree) {
13     this.tree = tree;
14     setStatus("Tree is empty");
15   }
16
17   public void setStatus(String msg) {
18     getChildren().add(new Text(20, 20, msg));
19   }
20
21   public void displayTree() {
22     this.getChildren().clear(); // Clear the pane
23     if (tree.getRoot() != null) {
24       // Display tree recursively
25       displayTree(tree.getRoot(), getWidth() / 2, vGap,
26         getWidth() / 4);
27     }
28   }
29
30   /** Display a subtree rooted at position (x, y) */
31   private void displayTree(BST.TreeNode<Integer> root,
32       double x, double y, double hGap) {
33     if (root.left != null) {
34       // Draw a line to the left node
35       getChildren().add(new Line(x - hGap, y + vGap, x, y));
36       // Draw the left subtree recursively
37       displayTree(root.left, x - hGap, y + vGap, hGap / 2);
38     }
39
40     if (root.right != null) {
41       // Draw a line to the right node
42       getChildren().add(new Line(x + hGap, y + vGap, x, y));
43       // Draw the right subtree recursively
44       displayTree(root.right, x + hGap, y + vGap, hGap / 2);
45     }
46
47     // Display a node
48     Circle circle = new Circle(x, y, radius);
49     circle.setFill(Color.WHITE);
50     circle.setStroke(Color.BLACK);
51     getChildren().addAll(circle,
52       new Text(x - 4, y + 4, root.element + ""));
53   }
54 }
```

In Listing 25.8, BSTAnimation.java, a tree is created (line 14) and a tree view is placed in the pane (line 18). After a new key is inserted into the tree (line 38), the tree is repainted (line 39) to reflect the change. After a key is deleted (line 51), the tree is repainted (line 52) to reflect the change.

In Listing 25.9, BTView.java, the node is displayed as a circle with `radius 15` (line 48). The distance between two levels in the tree is defined in `vGap 50` (line 25). `hGap` (line 32) defines the distance between two nodes horizontally. This value is reduced by half (`hGap / 2`) in the next level when the `displayTree` method is called recursively (lines 37 and 44). Note that `vGap` is not changed in the tree.

The method `displayTree` is recursively invoked to display a left subtree (lines 33–38) and a right subtree (lines 40–45) if a subtree is not empty. A line is added to the pane to connect two nodes (lines 35 and 42). Note the method first adds the lines to the pane then adds the circle into the pane (line 52) so the circles will be painted on top of the lines to achieve desired visual effects.

The program assumes the keys are integers. You can easily modify the program with a generic type to display keys of characters or short strings.

Tree visualization is an example of the model-view-controller (MVC) software architecture. This is an important architecture for software development. The model is for storing and handling data. The view is for visually presenting the data. The controller handles the user interaction with the model and controls the view, as shown in Figure 25.16.

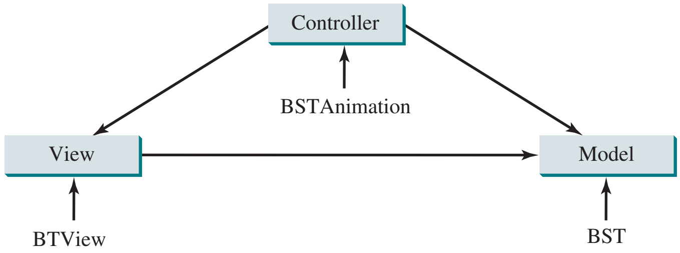


FIGURE 25.17 The controller obtains data and stores it in a model. The view displays the data stored in the model.

The MVC architecture separates data storage and handling from the visual representation of the data. It has two major benefits:

- It makes multiple views possible so data can be shared through the same model. For example, you can create a new view that displays the tree with the root on the left and the tree grows horizontally to the right (see Programming Exercise 25.11).
- It simplifies the task of writing complex applications and makes the components scalable and easy to maintain. Changes can be made to the view without affecting the model, and vice versa.

Check Point

25.9.1 How many times will the `displayTree` method be invoked if the tree is empty? How many times will the `displayTree` method be invoked if the tree has `100` nodes?

25.9.2 In what order are the nodes in the tree visited by the `displayTree` method: inorder, preorder, or postorder?

25.9.3 What would happen if the code in lines 47–52 in Listing 25.9, BTView.java is moved to line 33?

25.9.4 What is MVC? What are the benefits of the MVC?

25.10 Iterators

Key Point

`BST` is iterable because it is defined as a subtype of the `java.lang.Iterable` interface.

iterator

The methods `inorder()`, `preorder()`, and `postorder()` display the elements in `inorder`, `preorder`, and `postorder` in a binary tree. These methods are limited to displaying the elements in a tree. If you wish to process the elements in a binary tree rather than display them, these methods cannot be used. Recall that an iterator is provided for traversing the elements in a set or list. You can apply the same approach in a binary tree to provide a uniform way of traversing the elements in a binary tree.

The `java.lang.Iterable` interface defines the `iterator` method, which returns an instance of the `java.util.Iterator` interface. The `java.util.Iterator` interface (see Figure 25.17) defines the common features of iterators.

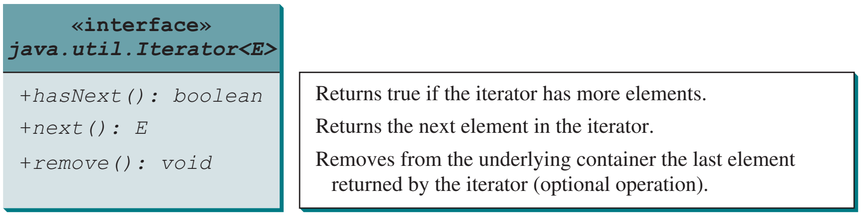

«interface»
java.util.Iterator<E>
+hasNext(): boolean
+next(): E
+remove(): void
Returns true if the iterator has more elements.
Returns the next element in the iterator.
Removes from the underlying container the last element returned by the iterator (optional operation).

Figure 25.18 The `Iterator` interface defines a uniform way of traversing the elements in a container.

The `Tree` interface extends `java.util.Collection`. Since `Collection` extends `java.lang.Iterable`, `BST` is also a subclass of `Iterable`. The `Iterable` interface contains the `iterator()` method that returns an instance of `java.util.Iterator`.

how to create an iterator

You can traverse a binary tree in inorder, preorder, or postorder. Since inorder is used frequently, we will use inorder for traversing the elements in a binary tree. We define an iterator class named `InorderIterator` to implement the `java.util.Iterator` interface in Listing 25.4 (lines 228–273). The `iterator` method simply returns an instance of `InorderIterator` (line 224).

The `InorderIterator` constructor invokes the `inorder` method (line 240). The `inorder(root)` method (lines 239–249) stores all the elements from the tree in `list`. The elements are traversed in `inorder`.

Once an `Iterator` object is created, its `current` value is initialized to `0` (line 232), which points to the first element in the list. Invoking the `next()` method returns the current element and moves `current` to point to the next element in the list (line 261).

The `hasNext()` method checks whether `current` is still in the range of `list` (line 253).

The `remove()` method removes the element returned by the last `next()` (line 269). Afterward, a new list is created (lines 270–271). Note that `current` does not need to be changed.

Listing 25.10 gives a test program that stores the strings in a BST and displays all strings in uppercase.

Listing 25.10 TestBSTWithIterator.java

use an iterator
get uppercase letters

```
 1 public class TestBSTWithIterator {
 2   public static void main(String[] args) {
 3     BST<String> tree = new BST<>();
 4     tree.insert("George");
 5     tree.insert("Michael");
 6     tree.insert("Tom");
 7     tree.insert("Adam");
 8     tree.insert("Jones");
 9     tree.insert("Peter");
10     tree.insert("Daniel");
11 
12     for (String s: tree)
13       System.out.print(s.toUpperCase() + " ");
14   }
15 }
```

```
ADAM DANIEL GEORGE JONES MICHAEL PETER TOM
```

The foreach loop (lines 12 and 13) uses an iterator to traverse all elements in the tree.

iterator pattern
advantages of iterators

Design Guide

Iterator is an important software design pattern. It provides a uniform way of traversing the elements in a container, while hiding the container's structural details. By implementing the same interface `java.util.Iterator`, you can write a program that traverses the elements of all containers in the same way.

Note

`java.util.Iterator` defines a forward iterator, which traverses the elements in the iterator in a forward direction, and each element can be traversed only once. The Java API also provides the `java.util.ListIterator`, which supports traversing in both forward and backward directions. If your data structure warrants flexible traversing, you may define iterator classes as a subtype of `java.util.ListIterator`.

variations of iterators

The implementation of the iterator is not efficient. Every time you remove an element through the iterator, the whole list is rebuilt (lines 270-271 in Listing 25.4, BST.java). The client should always use the `delete` method in the `BST` class to remove an element. To prevent the user from using the `remove` method in the iterator, implement the iterator as follows:

```
public void remove() {
  throw new UnsupportedOperationException
    ("Removing an element from the iterator is not supported");
}
```

After making the `remove` method unsupported by the iterator class, you can implement the iterator more efficiently without having to maintain a list for the elements in the tree. You can use a stack to store the nodes, and the node on the top of the stack contains the element that is to be returned from the `next()` method. If the tree is well balanced, the maximum stack size will be $O(\log n)$.

Check Point

25.10.1 What is an iterator?

25.10.2 What method is defined in the `java.lang.Iterable<E>` interface?

25.10.3 Suppose you delete `implements Collection<E>` from line 3 in Listing 25.3, Tree.java. Will Listing 25.10 still compile?

25.10.4 What is the benefit of being a subtype of `Iterable<E>`?

25.10.5 Write one statement that displays the maximum and minimum element in a `BST` object named `tree` (*Hint*: use the `min` and `max` methods in the `java.util.Collections` class).

25.11 Case Study: Data Compression

Key Point

Huffman coding compresses data by using fewer bits to encode characters that occur more frequently. The codes for the characters are constructed based on the occurrence of the characters in the text using a binary tree, called the Huffman coding tree.

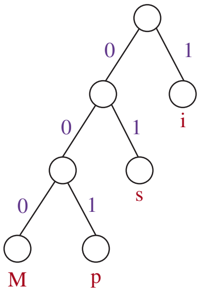
Compressing data is a common task. There are many utilities available for compressing files. This section introduces Huffman coding, invented by David Huffman in 1952.

In ASCII, every character is encoded in 8 bits. If a text consists of 100 characters, it will take 800 bits to represent the text. The idea of Huffman coding is to use a fewer bits to encode frequently used characters in the text and more bits to encode less frequently used characters to reduce the overall size of the file. In Huffman coding, the characters' codes are constructed based on the characters' occurrence in the text using a binary tree, called the *Huffman coding tree*. Suppose the text is `Mississippi`. Its Huffman tree is as shown in Figure 25.18a. The left and right edges of a node are assigned the values `0` and `1`, respectively. Each character is a leaf in the tree. The code for the character consists of the edge values in the path from the root to the leaf, as shown in Figure 25.18b. Since `i` and `s` appear more than `M` and `p` in the text, they are assigned shorter codes.

Huffman coding

The coding tree is also used for decoding a sequence of bits into characters. To do so, start with the first bit in the sequence and determine whether to go to the left or right branch of the tree's root based on the bit value. Consider the next bit and continue to go down to the left or right branch based on the bit value. When you reach a leaf, you have found a character. The

decoding



(a) Huffman coding tree

Character	Code	Frequency
M	000	1
p	001	2
s	01	4
i	1	4

(b) Character code table

FIGURE 25.19 The codes for characters are constructed based on the occurrence of characters in the text using a coding tree.

next bit in the stream is the first bit of the next character. For example, the stream `011001` is decoded to `sip`, with `01` matching `s`, `1` matching `i`, and `001` matching `p`.

Based on the coding scheme in Figure 25.19,

`is encoded to` `is decoded to`

Mississippi ==========> 000101011010110010011 ===========> Mississippi

construct coding tree

To construct a *Huffman coding tree*, use the following algorithm:

1. Begin with a forest of trees. Each tree contains a node for a character. The weight of the node is the frequency of the character in the text.
2. Repeat the following action to combine trees until there is only one tree: Choose two trees with the smallest weight and create a new node as their parent. The weight of the new tree is the sum of the weight of the subtrees.
3. For each interior node, assign its left edge a value `0` and right edge a value `1`. All leaf nodes represent characters in the text.

Here is an example of building a coding tree for the text `Mississippi`. The frequency table for the characters is shown in Figure 25.19b. Initially, the forest contains single-node trees, as shown in Figure 25.20a. The trees are repeatedly combined to form large trees until only one tree is left, as shown in Figures 25.20b–d.

prefix property

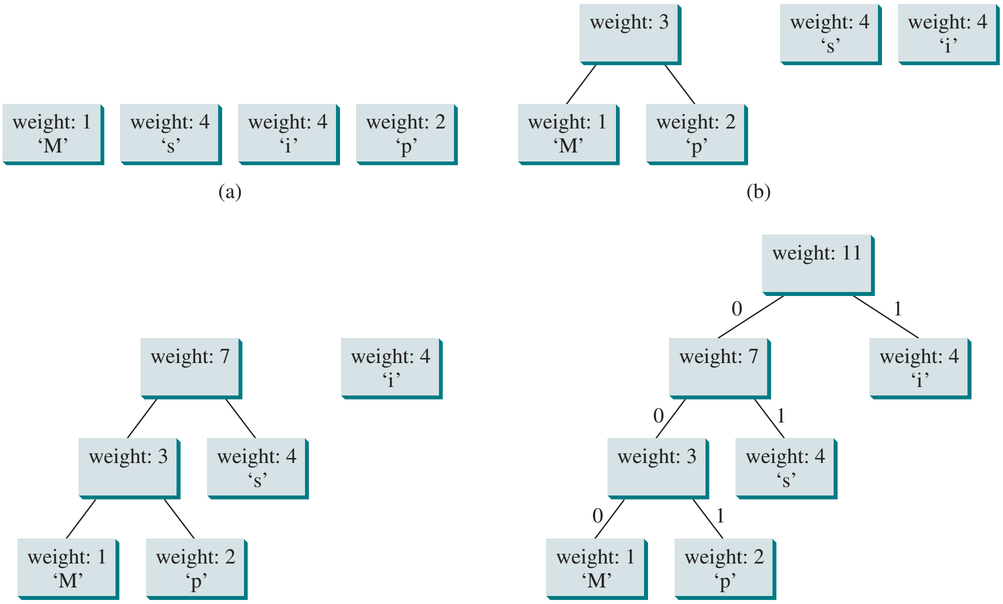
It is worth noting that no code is a prefix of another code. This property ensures that the streams can be decoded unambiguously.

greedy algorithm

Algorithm Design Note

The algorithm used here is an example of a *greedy algorithm*. A greedy algorithm is often used in solving optimization problems. The algorithm makes the choice that is optimal locally in the hope that this choice will lead to a globally optimal solution. In this case, the algorithm always chooses two trees with the smallest weight and creates a new node as their parent. This intuitive optimal local solution indeed leads to a final optimal solution for constructing a Huffman tree.

As another example of a greedy algorithm, consider changing money into the fewest possible coins. A greedy algorithm would take the largest possible coin first. For example, for 98¢, you would use three quarters to make 75¢, additional two dimes to make 95¢, and additional three pennies to make the 98¢. The greedy algorithm finds an optimal solution for this problem. However, a greedy algorithm is not always going to find the optimal result; see the bin packing problem in Programming Exercise 11.19.



(a) (b) (c) (d)

FIGURE 25.20 The coding tree is built by repeatedly combining the two smallest-weighted trees.

Listing 25.11 gives a program that prompts the user to enter a string, displays the frequency table of the characters in the string, and displays the Huffman code for each character.

LISTING 25.11 HuffmanCode.java

```
import java.util.Scanner;

public class HuffmanCode {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter text: ");
    String text = input.nextLine();

    int[] counts = getCharacterFrequency(text); // Count frequency

    System.out.printf("%-15s%-15s%-15s%-15s\n",
      "ASCII Code", "Character", "Frequency", "Code");

    Tree tree = getHuffmanTree(counts); // Create a Huffman tree
    String[] codes = getCode(tree.root); // Get codes

    for (int i = 0; i < codes.length; i++)
      if (counts[i] != 0) // (char)i is not in text if counts[i] is 0
        System.out.printf("%-15d%-15s%-15d%-15s\n",
          i, (char)i + "", counts[i], codes[i]);
  }

  /** Get Huffman codes for the characters
   * This method is called once after a Huffman tree is built
   */
```

count frequency

get Huffman tree

code for each character

getCode

```
  public static String[] getCode(Tree.Node root) {
    if (root == null)  return null;
    String[] codes = new String[128];
    assignCode(root, codes);
    return codes;
  }

  /* Recursively get codes to the leaf node */
```

assignCode

```
  private static void assignCode(Tree.Node root, String[] codes) {
    if (root.left != null) {
      root.left.code = root.code + "0";
      assignCode(root.left, codes);

      root.right.code = root.code + "1";
      assignCode(root.right, codes);
    }
    else {
      codes[(int)root.element] = root.code;
    }
  }

  /** Get a Huffman tree from the codes */
```

getHuffmanTree

```
  public static Tree getHuffmanTree(int[] counts) {
    // Create a heap to hold trees
    Heap<Tree> heap = new Heap<>(); // Defined in Listing 23.9
    for (int i = 0; i < counts.length; i++) {
      if (counts[i] > 0)
        heap.add(new Tree(counts[i], (char)i)); // A leaf node tree
    }

    while (heap.getSize() > 1) {
      Tree t1 = heap.remove(); // Remove the smallest-weight tree
      Tree t2 = heap.remove(); // Remove the next smallest
      heap.add(new Tree(t1, t2)); // Combine two trees
    }

    return heap.remove(); // The final tree
  }

  /** Get the frequency of the characters */
```

getCharacterFrequency

```
  public static int[] getCharacterFrequency(String text) {
    int[] counts = new int[128]; // 128 ASCII characters

    for (int i = 0; i < text.length(); i++)
      counts[(int)text.charAt(i)]++; // Count the characters in text

    return counts;
  }

  /** Define a Huffman coding tree */
```

Huffman tree

```
  public static class Tree implements Comparable<Tree> {
    Node root; // The root of the tree

    /** Create a tree with two subtrees */
    public Tree(Tree t1, Tree t2) {
      root = new Node();
      root.left = t1.root;
      root.right = t2.root;
      root.weight = t1.root.weight + t2.root.weight;
    }
```

```
 86
 87      /** Create a tree containing a leaf node */
 88      public Tree(int weight, char element) {
 89        root = new Node(weight, element);
 90      }
 91
 92      @Override /** Compare trees based on their weights */
 93      public int compareTo(Tree t) {
 94        if (root.weight < t.root.weight) // Purposely reverse the order
 95          return 1;
 96        else if (root.weight == t.root.weight)
 97          return 0;
 98        else
 99          return -1;
100      }
101
102      public class Node {
103        char element; // Stores the character for a leaf node
104        int weight; // weight of the subtree rooted at this node
105        Node left; // Reference to the left subtree
106        Node right; // Reference to the right subtree
107        String code = ""; // The code of this node from the root
108
109        /** Create an empty node */
110        public Node() {
111        }
112
113        /** Create a node with the specified weight and character */
114        public Node(int weight, char element) {
115          this.weight = weight;
116          this.element = element;
117        }
118      }
119    }
120  }
```

tree node

```
Enter text: Welcome ↵Enter

ASCII Code   Character   Frequency   Code
87           W           1           110
99           c           1           111
101          e           2           10
108          l           1           011
109          m           1           010
111          o           1           00
```

The program prompts the user to enter a text string (lines 5–7) and counts the frequency of the characters in the text (line 9). The `getCharacterFrequency` method (lines 66–73) creates an array `counts` to count the occurrences of each of the 128 ASCII characters in the text. If a character appears in the text, its corresponding count is increased by `1` (line 70).

getCharacterFrequency

The program obtains a Huffman coding tree based on `counts` (line 14). The tree consists of linked nodes. The `Node` class is defined in lines 102–118. Each node consists of properties `element` (storing character), `weight` (storing weight of the subtree under this node), `left` (linking to the left subtree), `right` (linking to the right subtree), and `code` (storing the Huffman code for the character). The `Tree` class (lines 76–119) contains the

Node class

`Tree` class

root property. From the root, you can access all the nodes in the tree. The `Tree` class implements `Comparable`. The trees are comparable based on their weights. The compare order is purposely reversed (lines 93–100) so the smallest-weight tree is removed first from the heap of trees.

`getHuffmanTree`

The `getHuffmanTree` method returns a Huffman coding tree. Initially, the single-node trees are created and added to the heap (lines 50–54). In each iteration of the `while` loop (lines 56–60), two smallest-weight trees are removed from the heap and are combined to form a big tree, then the new tree is added to the heap. This process continues until the heap contains just one tree, which is our final Huffman tree for the text.

`assignCode`
`getCode`

The `assignCode` method assigns the code for each node in the tree (lines 34–45). The `getCode` method gets the code for each character in the leaf node (lines 26–31). The element `codes[i]` contains the code for character `(char)i`, where `i` is from `0` to `127`. Note `codes[i]` is `null` if `(char)i` is not in the text.

Check Point

25.11.1 Every internal node in a Huffman tree has two children. Is it true?

25.11.2 What is a greedy algorithm? Give an example.

25.11.3 If the `Heap` class in line 50 in Listing 25.9 is replaced by `java.util.PriorityQueue`, will the program still work?

25.11.4 How do you replace lines 94–99 in Listing 25.11 using one line?

KEY TERMS

binary search tree	960
binary tree	960
breadth-first traversal	964
depth	960
depth-first traversal	964
greedy algorithm	986
height	960
Huffman coding	985
inorder traversal	963
leaf	960
length	960
level	960
postorder traversal	964
preorder traversal	964
sibling	960
tree traversal	963

CHAPTER SUMMARY

1. A *binary search tree* (BST) is a hierarchical data structure. You learned how to define and implement a BST class, how to insert and delete elements into/from a BST, and how to traverse a BST using *inorder*, *postorder*, *preorder*, depth-first, and breadth-first searches.

2. An iterator is an object that provides a uniform way of traversing the elements in a container, such as a set, a list, or a *binary tree*. You learned how to define and implement iterator classes for traversing the elements in a binary tree.

3. *Huffman coding* is a scheme for compressing data by using fewer bits to encode characters that occur more frequently. The codes for characters are constructed based on the occurrence of characters in the text using a binary tree, called the *Huffman coding tree*.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

Sections 25.2–25.6

***25.1** (*Tree height*) Define a new class named `BSTWithHeight` that extends `BST` with the following method:

```
/** Return the height of this binary tree */
public int height()
```

Use https://liveexample.pearsoncmg.com/test/Exercise25_01.txt to test your code.

****25.2** (*Implement inorder traversal without using recursion*) Implement the `inorder` method in `BST` using a stack instead of recursion. Write a test program that prompts the user to enter 10 integers, stores them in a BST, and invokes the `inorder` method to display the elements.

***25.3** (*Test perfect binary tree*) A perfect binary tree is a complete binary tree with all levels fully filled. Define a new class named `BSTWithTestPerfect` that extends `BST` with the following methods: (*Hint*: The number of nodes in a perfect binary tree is $2^{\text{height} + 1} - 1$).

```
/** Returns true if the tree is a perfect binary tree */
public boolean isPerfectBST()
```

Use https://liveexample.pearsoncmg.com/test/Exercise25_03.txt to test your code.

****25.4** (*Implement preorder traversal without using recursion*) Implement the `preorder` method in `BST` using a stack instead of recursion. Write a test program that prompts the user to enter 10 integers, stores them in a BST, and invokes the `preorder` method to display the elements.

****25.5** (*Implement postorder traversal without using recursion*) Implement the `postorder` method in `BST` using a stack instead of recursion. Write a test program that prompts the user to enter 10 integers, stores them in a BST, and invokes the `postorder` method to display the elements.

****25.6** (*Find the leaves*) Define a new class named `BSTWithNumberOfLeaves` that extends `BST` with the following methods:

```
/** Return the number of leaf nodes */
public int getNumberOfLeaves()
```

Use https://liveexample.pearsoncmg.com/test/Exercise25_06.txt to test your code.

****25.7** (*Find the nonleaves*) Define a new class named `BSTWithNumberOfNonLeaves` that extends `BST` with the following methods:

```
/** Return the number of nonleaf nodes */
public int getNumberofNonLeaves()
```

Use https://liveexample.pearsoncmg.com/test/Exercise25_07.txt to test your code.

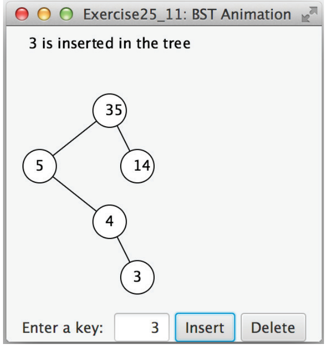
*****25.8** (*Implement bidirectional iterator*) The `java.util.Iterator` interface defines a forward iterator. The Java API also provides the `java.util.ListIterator` interface that defines a bidirectional iterator. Study `ListIterator` and define a bidirectional iterator for the `BST` class.

****25.9** (*Tree `clone` and `equals`*) Implement the `clone` and `equals` methods in the `BST` class. Two `BST` trees are equal if they contain the same elements. The `clone` method returns an identical copy of a `BST`.

25.10 (*Preorder iterator*) Add the following method in the `BST` class that returns an iterator for traversing the elements in a BST in preorder.

```
/** Return an iterator for traversing the elements in preorder */
java.util.Iterator<E> preorderIterator()
```

25.11 (*Display tree*) Write a new view class that displays the tree horizontally with the root on the left as shown in Figure 25.21.


Exercise25_11: BST Animation
35
5
14
4
3
Enter a key:
3
Insert
Delete

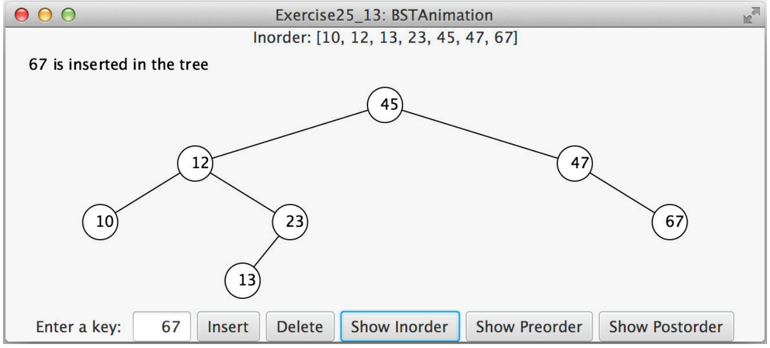
3 is inserted in the tree

FIGURE 25.21 A binary tree is displayed horizontally.

****25.12** (*Test `BST`*) Design and write a complete test program to test if the `BST` class in Listing 25.4 meets all requirements.

****25.13** (*Add new buttons in `BSTAnimation`*) Modify Listing 25.8, BSTAnimation.java, to add three new buttons—*Show Inorder, Show Preorder, and Show Postorder*—to display the result in a label, as shown in Figure 25.22. You need also to modify Listing 25.4, BST.java to implement the `inorderList()`, `preorderList()`, and `postorderList()` methods so each of these methods returns a `List` of the node elements in inorder, preorder, and postorder, as follows:

```
public java.util.List<E> inorderList();
public java.util.List<E> preorderList();
public java.util.List<E> postorderList();
```


Exercise25_13: BSTAnimation
45
12
47
10
23
67
13
Enter a key:
67
Insert
Delete
Show Inorder
Show Preorder
Show Postorder

Inorder: [10, 12, 13, 23, 45, 47, 67]

67 is inserted in the tree

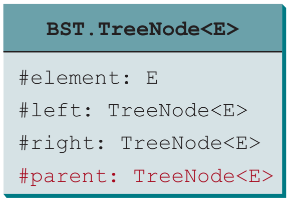
FIGURE 25.22 When you click the Show Inorder, Show Preorder, or Show Postorder button, the elements are displayed in an inorder, preorder, or postorder in a label. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***25.14** (*Breadth-first traversal*) Define a new class named `BSTWithBFT` that extends `BST` with the following method:

```
/** Display the nodes in a breadth-first traversal */
public void breadthFirstTraversal()
```

Use https://liveexample.pearsoncmg.com/test/Exercise25_14.txt to test your code.

*****25.15** (*Parent reference for `BST`*) Redefine `TreeNode` by adding a reference to a node's parent, as shown below:



Reimplement the `insert` and `delete` methods in the `BST` class to update the parent for each node in the tree. Add the following new method in `BST`:

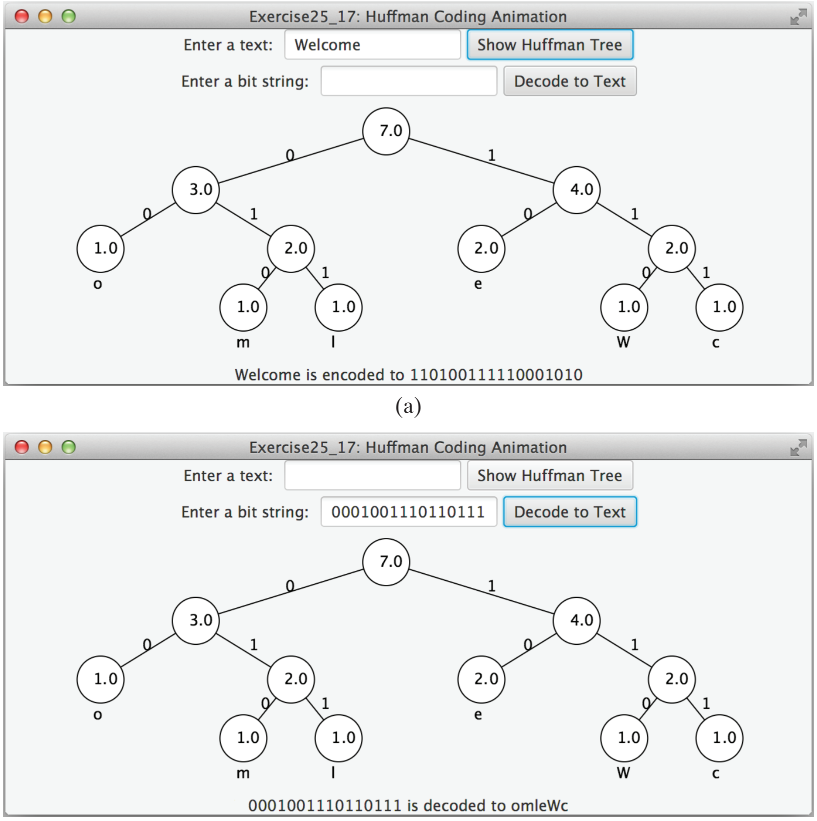
```
/** Return the node for the specified element.
  * Return null if the element is not in the tree. */
private TreeNode<E> getNode(E element)

/** Return true if the node for the element is a leaf */
private boolean isLeaf(E element)

/** Return the path of elements from the specified element
  * to the root in an array list. */
public ArrayList<E> getPath(E e)
```

Write a test program that prompts the user to enter 10 integers, adds them to the tree, deletes the first integer from the tree, and displays the paths for all leaf nodes. Here is a sample run:

```
Enter 10 integers: 45 54 67 56 50 45 23 59 23 67 ↵Enter
[50, 54, 23]
[59, 56, 67, 54, 23]
```

*****25.16** (*Data compression: Huffman coding*) Write a program that prompts the user to enter a file name, then displays the frequency table of the characters in the file and the Huffman code for each character.

*****25.17** (*Data compression: Huffman coding animation*) Write a program that enables the user to enter text and displays the Huffman coding tree based on the text, as shown in Figure 25.23a. Display the weight of the subtree inside the subtree's root circle. Display each leaf node's character. Display the encoded bits for the text in a label. When the user clicks the *Decode Text* button, a bit string is decoded into text displayed in the label, as shown in Figure 25.23b.



Enter a text: Welcome

Enter a bit string:

Welcome is encoded to 1101001111110001010

(a)

Enter a text:

Enter a bit string: 0001001110110111

0001001110110111 is decoded to omleWc

(b)

FIGURE 25.23 (a) The animation shows the coding tree for a given text string and the encoded bits for the text are displayed in the label; (b) You can enter a bit string to display its text in the label. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

*****25.18** (*Compress a file*) Write a program that compresses a source file into a target file using the Huffman coding method. First, use `ObjectOutputStream` to output the Huffman codes into the target file, then use `BitOutputStream` in Programming Exercise 17.17 to output the encoded binary contents to the target file. Pass the files from the command line using the following command:

```
java Exercise25_18 sourcefile targetfile
```

*****25.19** (*Decompress a file*) The preceding exercise compresses a file. The compressed file contains the Huffman codes and the compressed contents. Write a program that decompresses a source file into a target file using the following command:

```
java Exercise25_19 sourcefile targetfile
```

CHAPTER
26

AVL TREES

Objectives

- To know what an AVL tree is (§26.1).
- To understand how to rebalance a tree using the LL rotation, LR rotation, RR rotation, and RL rotation (§26.2).
- To design the `AVLTree` class by extending the `BST` class (§26.3).
- To insert elements into an AVL tree (§26.4).
- To implement tree rebalancing (§26.5).
- To delete elements from an AVL tree (§26.6).
- To implement the `AVLTree` class (§26.7).
- To test the `AVLTree` class (§26.8).
- To analyze the complexity of search, insertion, and deletion operations in AVL trees (§26.9).

26.1 Introduction

AVL Tree is a balanced binary search tree.

Key Point

perfectly balanced tree
well-balanced tree

Chapter 25 introduced binary search trees. The search, insertion, and deletion times for a binary tree depend on the height of the tree. In the worst case, the height is $O(n)$. If a tree is *perfectly balanced*—that is, a complete binary tree—its height is $\log n$. Can we maintain a perfectly balanced tree? Yes, but doing so will be costly. The compromise is to maintain a *well-balanced tree*—that is, the heights of every node's two subtrees are about the same. This chapter introduces AVL trees. Web Chapters 40 and 41 will introduce 2–4 trees and red–black trees.

AVL tree

AVL trees are well balanced. AVL trees were invented in 1962 by two Russian computer scientists, G. M. Adelson-Velsky and E. M. Landis (hence the name *AVL*). In an AVL tree, the difference between the heights of every node's two subtrees is **0** or **1**. It can be shown that the maximum height of an AVL tree is $O(\log n)$.

$O(\log n)$

The process for inserting or deleting an element in an AVL tree is the same as in a binary search tree, except that you may have to rebalance the tree after an insertion or deletion operation. The *balance factor* of a node is the height of its right subtree minus the height of its left subtree. For example, the balance factor for the node 87 in Figure 26.1a is **0**, for the node 67 is **1**, and for the node 55 is **−1**. A node is said to be *balanced* if its balance factor is **−1**, **0**, or **1**. A node is considered *left-heavy* if its balance factor is **−1** or less, and *right-heavy* if its balance factor is **+1** or greater.

balance factor
balanced
left-heavy
right-heavy

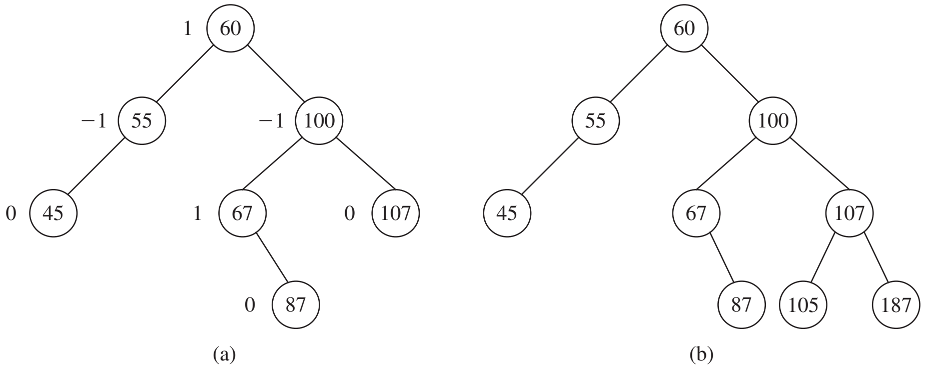


FIGURE 26.1 A balance factor determines whether a node is balanced.

AVL tree animation on Companion Website

Pedagogical Note

For an interactive GUI demo to see how an AVL tree works, go to liveexample.pearsoncmg.com/dsanimation/AVLTreeeBook.html, as shown in Figure 26.2.

26.2 Rebalancing Trees

After inserting or deleting an element from an AVL tree, if the tree becomes unbalanced, perform a rotation operation to rebalance the tree.

Key Point

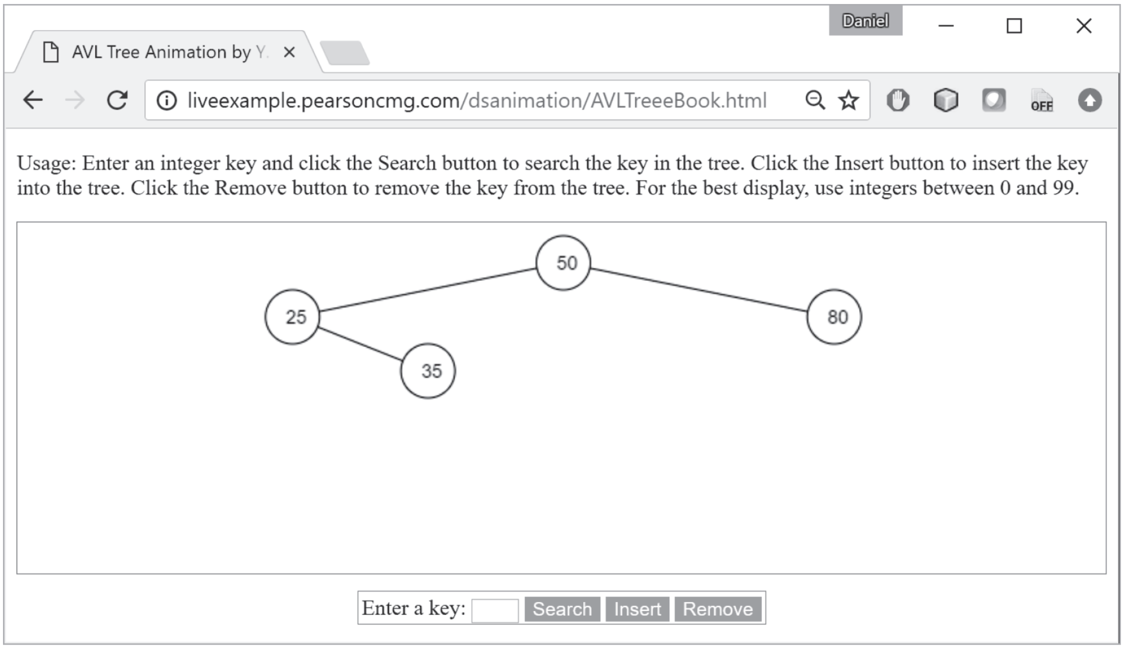
rotation
LL rotation
LL imbalance

If a node is not balanced after an insertion or deletion operation, you need to rebalance it. The process of rebalancing a node is called *rotation*. There are four possible rotations: LL, RR, LR, and RL.

LL rotation: An *LL imbalance* occurs at a node **A**, such that **A** has a balance factor of **−2** and a left child **B** with a balance factor of **−1** or **0**, as shown in Figure 26.3a. This type of imbalance can be fixed by performing a single right rotation at **A**, as shown in Figure 26.3b.

RR rotation
RR imbalance

RR rotation: An *RR imbalance* occurs at a node **A**, such that **A** has a balance factor of **+2** and a right child **B** with a balance factor of **+1** or **0**, as shown in Figure 26.4a. This type of imbalance can be fixed by performing a single left rotation at **A**, as shown in Figure 26.4b.


Daniel
AVL Tree Animation by Y. ×
liveexample.pearsoncmg.com/dsanimation/AVLTreeeBook.html
50
25
80
35
Enter a key:
Search
Insert
Remove

Usage: Enter an integer key and click the Search button to search the key in the tree. Click the Insert button to insert the key into the tree. Click the Remove button to remove the key from the tree. For the best display, use integers between 0 and 99.

FIGURE 26.2 The animation tool enables you to insert, delete, and search elements. *Source:* Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

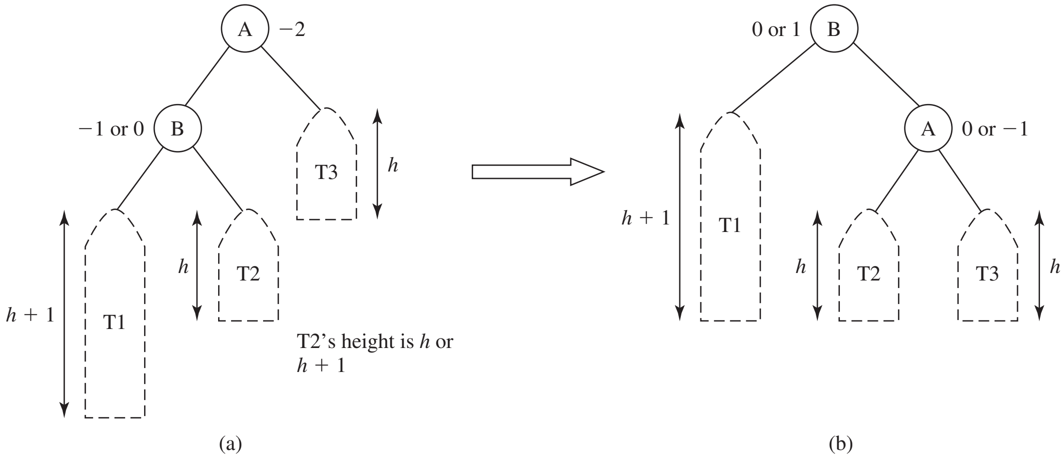

A
−2
B
−1 or 0
T3
h
T1
h + 1
T2
h
T2's height is h or h + 1
(a)
B
0 or 1
A
0 or −1
T1
h + 1
T2
h
T3
h
(b)

FIGURE 26.3 An LL rotation fixes an LL imbalance.

LR rotation: An *LR imbalance* occurs at a node **A**, such that **A** has a balance factor of **−2** and a left child **B** with a balance factor of **+1**, as shown in Figure 26.5a. Assume **B**'s right child is **C**. This type of imbalance can be fixed by performing a double rotation (first a single left rotation at **B**, then a single right rotation at **A**), as shown in Figure 26.5b.

LR rotation
LR imbalance

RL rotation: An *RL imbalance* occurs at a node **A**, such that **A** has a balance factor of **+2** and a right child **B** with a balance factor of **−1**, as shown in Figure 26.6a. Assume **B**'s left child is **C**. This type of imbalance can be fixed by performing a double rotation (first a single right rotation at **B**, then a single left rotation at **A**), as shown in Figure 26.6b.

RL rotation
RL imbalance

Check Point

26.2.1 What is an AVL tree? Describe the following terms: balance factor, left-heavy, and right-heavy.

26.2.2 Show the balance factor of each node in the trees shown in Figure 26.1.

26.2.3 Describe LL rotation, RR rotation, LR rotation, and RL rotation for an AVL tree.

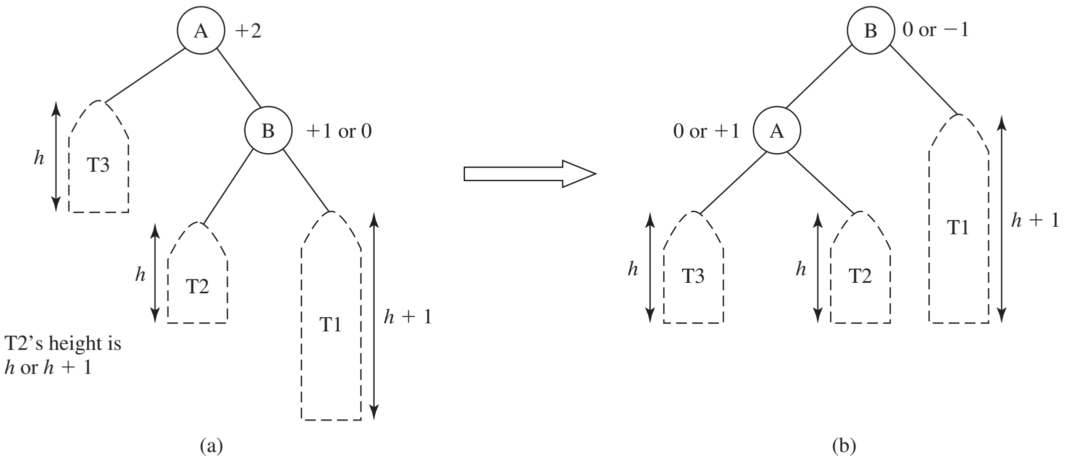

A
+2
B
+1 or 0
h
T3
h
T2
T1
h + 1
T2's height is
h or h + 1
(a)
B
0 or −1
0 or +1
A
h
T3
h
T2
T1
h + 1
(b)

FIGURE 26.4 An RR rotation fixes an RR imbalance.

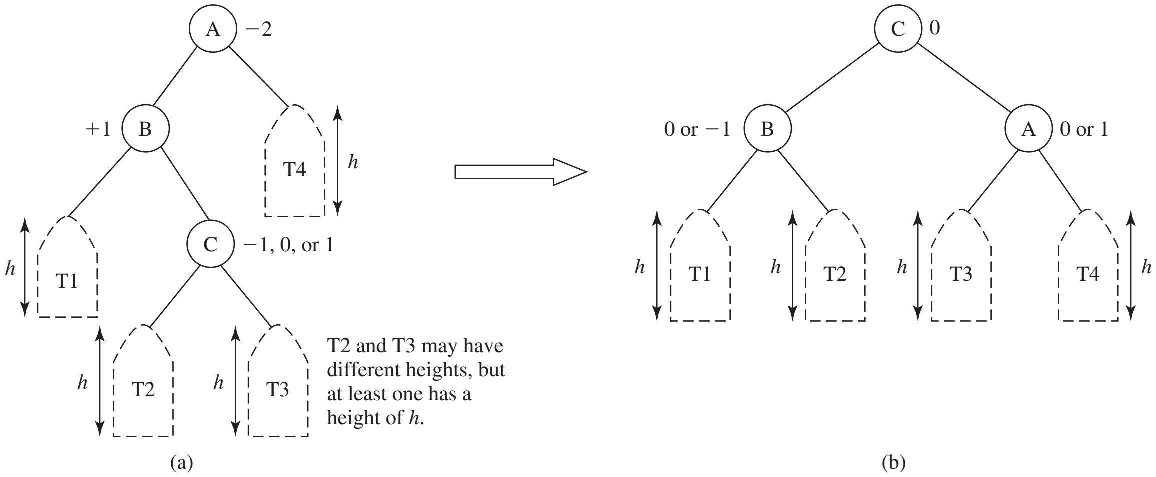

A
−2
+1
B
T4
h
h
T1
C
−1, 0, or 1
h
T2
h
T3
T2 and T3 may have
different heights, but
at least one has a
height of h.
(a)
C
0
0 or −1
B
A
0 or 1
h
T1
h
T2
h
T3
T4
h
(b)

FIGURE 26.5 An LR rotation fixes an LR imbalance.

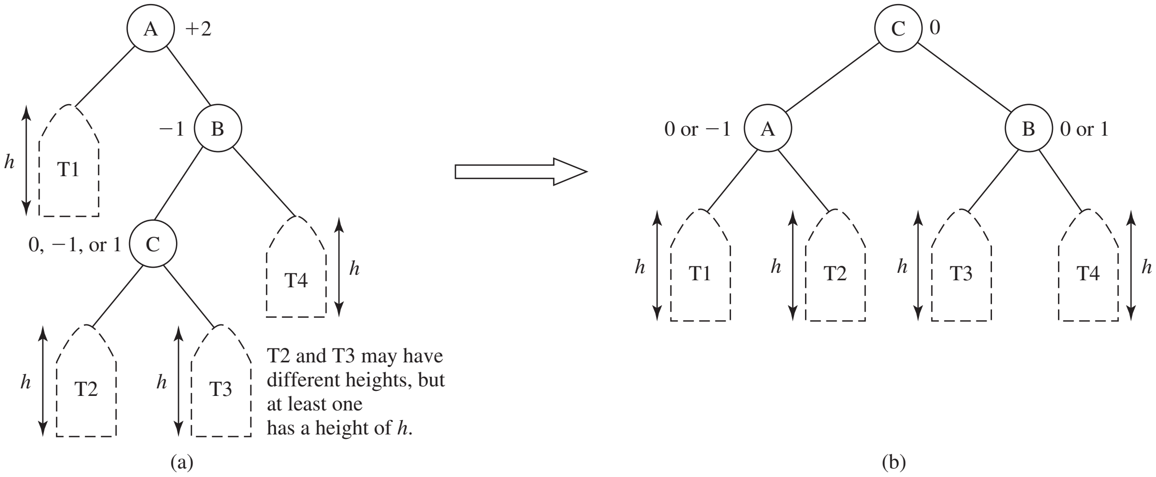

A
+2
h
T1
−1
B
0, −1, or 1
C
T4
h
h
T2
h
T3
T2 and T3 may have
different heights, but
at least one
has a height of h.
(a)
C
0
0 or −1
A
B
0 or 1
h
T1
h
T2
h
T3
T4
h
(b)

FIGURE 26.6 An RL rotation fixes an RL imbalance.

26.3 Designing Classes for AVL Trees

*Since an AVL tree is a binary search tree, **AVLTree** is designed as a subclass of **BST**.*

Key Point

An AVL tree is a binary tree, so you can define the **AVLTree** class to extend the **BST** class, as shown in Figure 26.7. The **BST** and **TreeNode** classes were defined in Section 25.2.5.

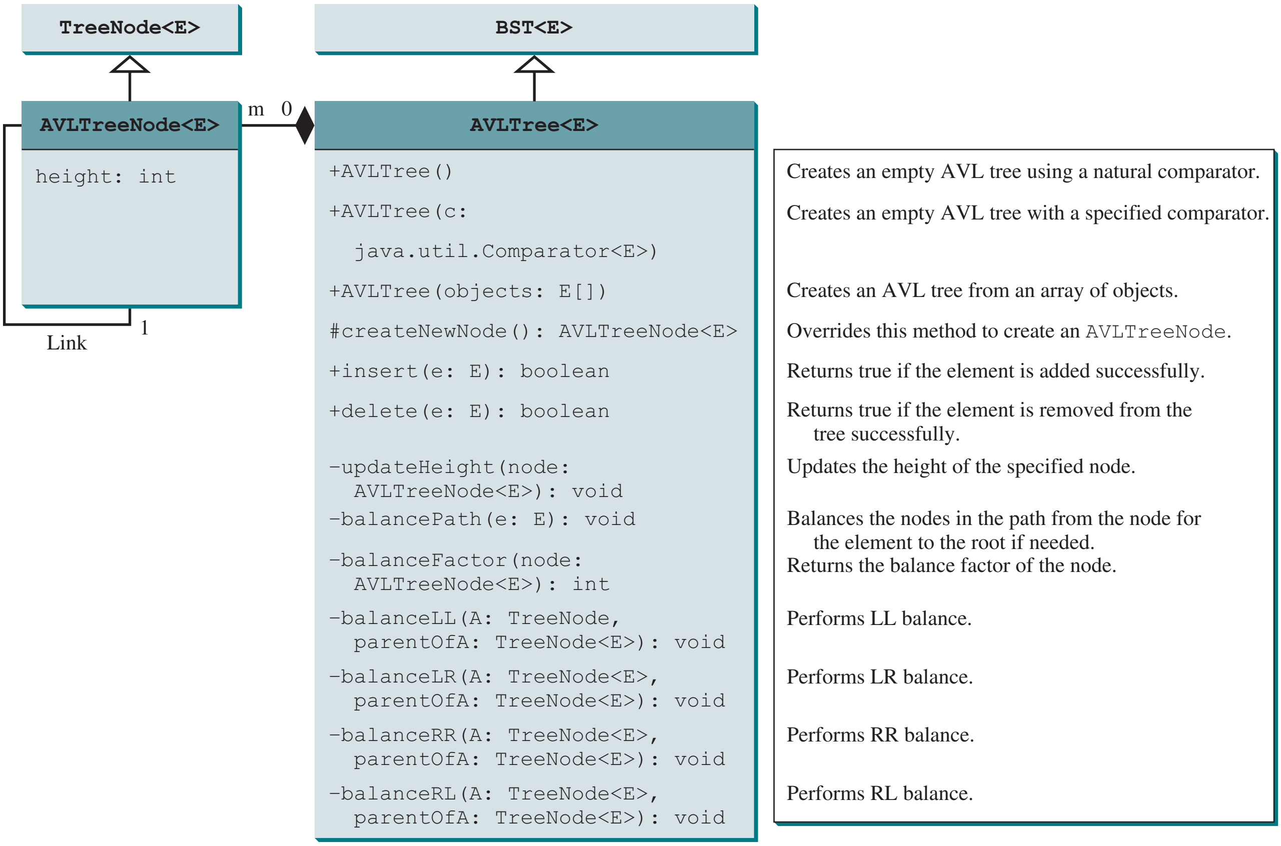

TreeNode<E>
BST<E>
AVLTreeNode<E>
height: int
m
0
1
Link
AVLTree<E>
+AVLTree()
+AVLTree(c: java.util.Comparator<E>)
+AVLTree(objects: E[])
#createNewNode(): AVLTreeNode<E>
+insert(e: E): boolean
+delete(e: E): boolean
-updateHeight(node: AVLTreeNode<E>): void
-balancePath(e: E): void
-balanceFactor(node: AVLTreeNode<E>): int
-balanceLL(A: TreeNode, parentOfA: TreeNode<E>): void
-balanceLR(A: TreeNode<E>, parentOfA: TreeNode<E>): void
-balanceRR(A: TreeNode<E>, parentOfA: TreeNode<E>): void
-balanceRL(A: TreeNode<E>, parentOfA: TreeNode<E>): void
Creates an empty AVL tree using a natural comparator.
Creates an empty AVL tree with a specified comparator.
Creates an AVL tree from an array of objects.
Overrides this method to create an AVLTreeNode.
Returns true if the element is added successfully.
Returns true if the element is removed from the tree successfully.
Updates the height of the specified node.
Balances the nodes in the path from the node for the element to the root if needed.
Returns the balance factor of the node.
Performs LL balance.
Performs LR balance.
Performs RR balance.
Performs RL balance.

FIGURE 26.7 The **AVLTree** class extends **BST** with new implementations for the **insert** and **delete** methods.

In order to balance the tree, you need to know each node's height. For convenience, store the height of each node in **AVLTreeNode** and define **AVLTreeNode** to be a subclass of **BST.TreeNode**. Note that **TreeNode** is defined as a static inner class in **BST**. **AVLTreeNode** will be defined as a static inner class in **AVLTree**. **TreeNode** contains the data fields **element**, **left**, and **right**, which are inherited by **AVLTreeNode**. Thus, **AVLTreeNode** contains four data fields, as shown in Figure 26.8.

AVLTreeNode

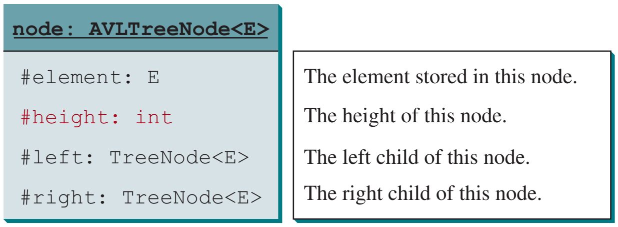

node: AVLTreeNode<E>
#element: E
#height: int
#left: TreeNode<E>
#right: TreeNode<E>
The element stored in this node.
The height of this node.
The left child of this node.
The right child of this node.

FIGURE 26.8 An **AVLTreeNode** contains the protected data fields **element**, **height**, **left**, and **right**.

createNewNode()

In the `BST` class, the `createNewNode()` method creates a `TreeNode` object. This method is overridden in the `AVLTree` class to create an `AVLTreeNode`. Note the return type of the `createNewNode()` method in the `BST` class is `TreeNode`, but the return type of the `createNewNode()` method in the `AVLTree` class is `AVLTreeNode`. This is fine, since `AVLTreeNode` is a subclass of `TreeNode`.

Searching for an element in an `AVLTree` is the same as searching in a binary search tree, so the `search` method defined in the `BST` class also works for `AVLTree`.

The `insert` and `delete` methods are overridden to insert and delete an element and perform rebalancing operations if necessary to ensure that the tree is balanced.

Check Point

26.3.1 What are the data fields in the `AVLTreeNode` class?

26.3.2 True or false: `AVLTreeNode` is a subclass of `TreeNode`.

26.3.3 True or false: `AVLTree` is a subclass of `BST`.

26.4 Overriding the `insert` Method

Key Point

Inserting an element into an AVL tree is the same as inserting it to a BST, except that the tree may need to be rebalanced.

A new element is always inserted as a leaf node. As a result of adding a new node, the heights of the new leaf node's ancestors may increase. After inserting a new node, check the nodes along the path from the new leaf node up to the root. If an unbalanced node is found, perform an appropriate rotation using the algorithm in Listing 26.1.

LISTING 26.1 Balancing Nodes on a Path

get the path

update node height

get parent node

is balanced?

LL rotation

LR rotation

RR rotation

RL rotation

```
balancePath(E e) {
  Get the path from the node that contains element e to the root,
    as illustrated in Figure 26.9;
  for each node A in the path leading to the root {
    Update the height of A;
    Let parentOfA denote the parent of A,
      which is the next node in the path, or null if A is the root;

    switch (balanceFactor(A)) {
      case -2:  if balanceFactor(A.left) == -1 or 0
                  Perform LL rotation; // See Figure 26.3
                else
                  Perform LR rotation; // See Figure 26.5
                break;
      case +2:  if balanceFactor(A.right) == +1 or 0
                  Perform RR rotation; // See Figure 26.4
                else
                  Perform RL rotation; // See Figure 26.6
    } // End of switch
  } // End of for
} // End of method
```

The algorithm considers each node in the path from the new leaf node to the root. Update the height of the node on the path. If a node is balanced, no action is needed. If a node is not balanced, perform an appropriate rotation.

Check Point

26.4.1 For the AVL tree in Figure 26.1a, show the new AVL tree after adding element `40`. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

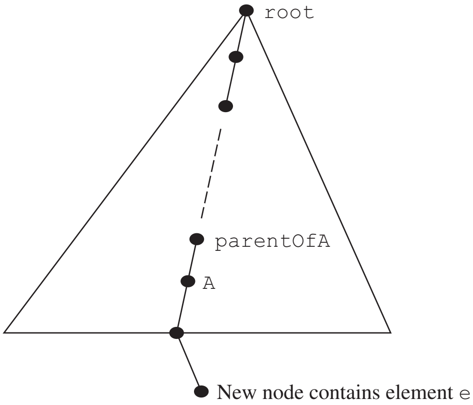


FIGURE 26.9 The nodes along the path from the new leaf node may become unbalanced.

26.4.2 For the AVL tree in Figure 26.1a, show the new AVL tree after adding element **50**. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

26.4.3 For the AVL tree in Figure 26.1a, show the new AVL tree after adding element **80**. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

26.4.4 For the AVL tree in Figure 26.1a, show the new AVL tree after adding element **89**. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

26.5 Implementing Rotations

An unbalanced tree becomes balanced by performing an appropriate rotation operation.

Section 26.2, Rebalancing Trees, illustrated how to perform rotations at a node. Listing 26.2 gives the algorithm for the LL rotation, as illustrated in Figure 26.3.

LISTING 26.2 LL Rotation Algorithm

```
balanceLL(TreeNode A, TreeNode parentOfA) {
  Let B be the left child of A.

  if (A is the root)
    Let B be the new root
  else {
    if (A is a left child of parentOfA)
      Let B be a left child of parentOfA;
    else
      Let B be a right child of parentOfA;
  }

  Make T2 the left subtree of A by assigning B.right to A.left;
  Make A the right child of B by assigning A to B.right;
  Update the height of node A and node B;
} // End of method
```

left child of A

reconnect B's parent

move subtrees

adjust height

Note the height of nodes **A** and **B** can be changed, but the heights of other nodes in the tree are not changed. You can implement the RR, LR, and RL rotations in a similar manner.

26.5.1 Use Listing 26.2 as a template to describe the algorithms for implementing the RR, LR, and RL rotations.

26.6 Implementing the `delete` Method

Key Point

Deleting an element from an AVL tree is the same as deleting it from a BST, except that the tree may need to be rebalanced.

As discussed in Section 25.3, Deleting Elements from a BST, to delete an element from a binary tree, the algorithm first locates the node that contains the element. Let `current` point to the node that contains the element in the binary tree and `parent` point to the parent of the `current` node. The `current` node may be a left child or a right child of the `parent` node. Two cases arise when deleting an element.

Case 1**:** The `current` node does not have a left child, as shown in Figure 25.10a. To delete the `current` node, simply connect the `parent` node with the right child of the `current` node, as shown in Figure 25.10b.

The height of the nodes along the path from the `parent` node up to the `root` may have decreased. To ensure that the tree is balanced, invoke

```
balancePath(parent.element); // Defined in Listing 26.1
```

Case 2**:** The `current` node has a left child. Let `rightMost` point to the node that contains the largest element in the left subtree of the `current` node and `parentOfRightMost` point to the parent node of the `rightMost` node, as shown in Figure 25.12a. The `rightMost` node cannot have a right child, but may have a left child. Replace the element value in the `current` node with the one in the `rightMost` node, connect the `parentOfRightMost` node with the left child of the `rightMost` node, and delete the `rightMost` node, as shown in Figure 25.12b.

The height of the nodes along the path from `parentOfRightMost` up to the root may have decreased. To ensure the tree is balanced, invoke

```
balancePath(parentOfRightMost); // Defined in Listing 26.1
```

Check Point

26.6.1 For the AVL tree in Figure 26.1a, show the new AVL tree after deleting element `107`. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

26.6.2 For the AVL tree in Figure 26.1a, show the new AVL tree after deleting element `60`. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

26.6.3 For the AVL tree in Figure 26.1a, show the new AVL tree after deleting element `55`. What rotation did you perform in order to rebalance the tree? Which node was unbalanced?

26.6.4 For the AVL tree in Figure 26.1b, show the new AVL tree after deleting elements `67` and `87`. What rotation did you perform in order to rebalance the tree? Which node was unbalanced?

26.7 The `AVLTree` Class

Key Point

The `AVLTree` class extends the `BST` class to override the `insert` and `delete` methods to rebalance the tree if necessary.

Listing 26.3 gives the complete source code for the `AVLTree` class.

Listing 26.3 `AVLTree.java`

extends BST

no-arg constructor

```
public class AVLTree<E> extends BST<E> {
  /** Create an empty AVL tree using a natural comparator*/
  public AVLTree() { // super() is implicitly called
  }

  /** Create a BST with a specified comparator */
  public AVLTree(java.util.Comparator<E> c) {
    super(c);
  }

  /** Create an AVL tree from an array of objects */
  public AVLTree(E[] objects) {
    super(objects);
  }

  @Override /** Override createNewNode to create an AVLTreeNode */
  protected AVLTreeNode<E> createNewNode(E e) {
    return new AVLTreeNode<E>(e);
  }

  @Override /** Insert an element and rebalance if necessary */
  public boolean insert(E e) {
    boolean successful = super.insert(e);
    if (!successful)
      return false; // e is already in the tree
    else {
      balancePath(e); // Balance from e to the root if necessary
    }

    return true; // e is inserted
  }

  /** Update the height of a specified node */
  private void updateHeight(AVLTreeNode<E> node) {
    if (node.left == null && node.right == null) // node is a leaf
      node.height = 0;
    else if (node.left == null) // node has no left subtree
      node.height = 1 + ((AVLTreeNode<E>)(node.right)).height;
    else if (node.right == null) // node has no right subtree
      node.height = 1 + ((AVLTreeNode<E>)(node.left)).height;
    else
      node.height = 1 +
        Math.max(((AVLTreeNode<E>)(node.right)).height,
        ((AVLTreeNode<E>)(node.left)).height);
  }

  /** Balance the nodes in the path from the specified
   * node to the root if necessary
   */
  private void balancePath(E e) {
    java.util.ArrayList<TreeNode<E>> path = path(e);
    for (int i = path.size() - 1; i >= 0; i--) {
      AVLTreeNode<E> A = (AVLTreeNode<E>)(path.get(i));
      updateHeight(A);
      AVLTreeNode<E> parentOfA = (A == root) ? null :
        (AVLTreeNode<E>)(path.get(i - 1));

      switch (balanceFactor(A)) {
        case -2:
          if (balanceFactor((AVLTreeNode<E>)A.left) <= 0) {
            balanceLL(A, parentOfA); // Perform LL rotation
          }
          else {
```

constructor with a comparator

constructor for arrays

create AVL tree node

override insert

balance tree

update node height

balance nodes

get path

consider a node

update height

get parentOfA

left-heavy

LL rotation

LR rotation
right-heavy
RR rotation
RL rotation
get balance factor
LL rotation
update height
LR rotation

```
          balanceLR(A, parentOfA); // Perform LR rotation
        }
        break;
      case +2:
        if (balanceFactor((AVLTreeNode<E>)A.right) >= 0) {
          balanceRR(A, parentOfA); // Perform RR rotation
        }
        else {
          balanceRL(A, parentOfA); // Perform RL rotation
        }
    }
  }
}

/** Return the balance factor of the node */
private int balanceFactor(AVLTreeNode<E> node) {
  if (node.right == null) // node has no right subtree
    return -node.height;
  else if (node.left == null) // node has no left subtree
    return +node.height;
  else
    return ((AVLTreeNode<E>)node.right).height -
      ((AVLTreeNode<E>)node.left).height;
}

/** Balance LL (see Figure 26.3) */
private void balanceLL(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.left; // A is left-heavy and B is left-heavy

  if (A == root) {
    root = B;
  }
  else {
    if (parentOfA.left == A) {
      parentOfA.left = B;
    }
    else {
      parentOfA.right = B;
    }
  }

  A.left = B.right; // Make T2 the left subtree of A
  B.right = A; // Make A the left child of B
  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B);
}

/** Balance LR (see Figure 26.5) */
private void balanceLR(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.left; // A is left-heavy
  TreeNode<E> C = B.right; // B is right-heavy

  if (A == root) {
    root = C;
  }
  else {
    if (parentOfA.left == A) {
      parentOfA.left = C;
    }
    else {
        parentOfA.right = C;
      }
    }

    A.left = C.right; // Make T3 the left subtree of A
    B.right = C.left; // Make T2 the right subtree of B
    C.left = B;
    C.right = A;

    // Adjust heights
    updateHeight((AVLTreeNode<E>)A);
    updateHeight((AVLTreeNode<E>)B);
    updateHeight((AVLTreeNode<E>)C);
  }

  /** Balance RR (see Figure 26.4) */
  private void balanceRR(TreeNode<E> A, TreeNode<E> parentOfA) {
   TreeNode<E> B = A.right; // A is right-heavy and B is right-heavy

    if (A == root) {
      root = B;
    }
    else {
      if (parentOfA.left == A) {
        parentOfA.left = B;
      }
      else {
        parentOfA.right = B;
      }
    }

    A.right = B.left; // Make T2 the right subtree of A
    B.left = A;
    updateHeight((AVLTreeNode<E>)A);
    updateHeight((AVLTreeNode<E>)B);
  }

  /** Balance RL (see Figure 26.6) */
  private void balanceRL(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.right; // A is right-heavy
    TreeNode<E> C = B.left; // B is left-heavy

    if (A == root) {
      root = C;
    }
    else {
      if (parentOfA.left == A) {
        parentOfA.left = C;
      }
      else {
        parentOfA.right = C;
      }
    }

    A.right = C.left; // Make T2 the right subtree of A
    B.left = C.right; // Make T3 the left subtree of B
    C.left = A;
    C.right = B;

    // Adjust heights
```

update height

RR rotation

update height

RL rotation

update height

```
      updateHeight((AVLTreeNode<E>)A);
      updateHeight((AVLTreeNode<E>)B);
      updateHeight((AVLTreeNode<E>)C);
    }

    @Override /** Delete an element from the binary tree.
     * Return true if the element is deleted successfully
     * Return false if the element is not in the tree */
```

override delete

```
    public boolean delete(E element) {
      if (root == null)
        return false; // Element is not in the tree

      // Locate the node to be deleted and also locate its parent node
      TreeNode<E> parent = null;
      TreeNode<E> current = root;
      while (current != null) {
        if (c.compare(element, current.element) < 0) {
          parent = current;
          current = current.left;
        }
        else if (c.compare(element, current.element) > 0) {
          parent = current;
          current = current.right;
        }
        else
          break; // Element is in the tree pointed by current
      }

      if (current == null)
        return false; // Element is not in the tree

      // Case 1: current has no left children (See Figure 23.6)
      if (current.left == null) {
        // Connect the parent with the right child of the current node
        if (parent == null) {
          root = current.right;
        }
        else {
          if (c.compare(element, parent.element) < 0)
            parent.left = current.right;
          else
            parent.right = current.right;

          // Balance the tree if necessary
```

balance nodes

```
          balancePath(parent.element);
        }
      }
      else {
        // Case 2: The current node has a left child
        // Locate the rightmost node in the left subtree of
        // the current node and also its parent
        TreeNode<E> parentOfRightMost = current;
        TreeNode<E> rightMost = current.left;

        while (rightMost.right != null) {
          parentOfRightMost = rightMost;
          rightMost = rightMost.right; // Keep going to the right
        }

       // Replace the element in current by the element in rightMost
```

```
244         current.element = rightMost.element;
245
246         // Eliminate rightmost node
247         if (parentOfRightMost.right == rightMost)
248           parentOfRightMost.right = rightMost.left;
249         else
250           // Special case: parentOfRightMost is current
251           parentOfRightMost.left = rightMost.left;
252
253         // Balance the tree if necessary
254         balancePath(parentOfRightMost.element);
255       }
256
257       size--;
258       return true; // Element inserted
259     }
260
261     /** AVLTreeNode is TreeNode plus height */
262     protected static class AVLTreeNode<E> extends BST.TreeNode<E> {
263       protected int height = 0; // New data field
264
265       public AVLTreeNode(E o) {
266         super(o);
267       }
268     }
269 }
```

balance nodes

inner AVLTreeNode class

node height

constructors

The **AVLTree** class extends **BST**. Like the **BST** class, the **AVLTree** class has a no-arg constructor that constructs an empty **AVLTree** (lines 3 and 4) using a natural comparator, a constructor that constructs an empty AVLTree (lines 7–9) with a specified comparator, and a constructor that creates an initial **AVLTree** from an array of elements (lines 12–14).

The **createNewNode()** method defined in the **BST** class creates a **TreeNode**. This method is overridden to return an **AVLTreeNode** (lines 17–19).

`insert`

The **insert** method in **AVLTree** is overridden in lines 22–31. The method first invokes the **insert** method in **BST**, then invokes **balancePath(e)** (line 27) to ensure that the tree is balanced.

`balancePath`

The **balancePath** method first gets the nodes on the path from the node that contains element **e** to the root (line 51). For each node in the path, update its height (line 58), check its balance factor (line 58), and perform appropriate rotations if necessary (lines 59–73).

rotations

Four methods for performing rotations are defined in lines 90–187. Each method is invoked with two **TreeNode** arguments—**A** and **parentOfA**—to perform an appropriate rotation at node **A**. How each rotation is performed is illustrated in Figures 26.3–26.6. After the rotation, the heights of nodes **A**, **B**, and **C** are updated (lines 107, 134, 157, and 184).

`delete`

The **delete** method in **AVLTree** is overridden in lines 192–259. The method is the same as the one implemented in the **BST** class, except that you have to rebalance the nodes after deletion in two cases (lines 228, 254).

Check Point

26.7.1 Why is the **createNewNode** method defined protected? When is it invoked?

26.7.2 When is the **updateHeight** method invoked? When is the **balanceFactor** method invoked? When is the **balancePath** method invoked? Will the program work if you replace the break in line 61 in the **AVLTree** class with a return and add a return at line 69?

26.7.3 What are data fields in the **AVLTree** class?

26.7.4 In the **insert** and **delete** methods, once you have performed a rotation to balance a node in the tree, is it possible there are still unbalanced nodes?

26.8 Testing the **AVLTree** Class

Key Point

This section gives an example of using the **AVLTree** *class.*

Listing 26.4 gives a test program. The program creates an **AVLTree** initialized with an array of the integers **25**, **20**, and **5** (lines 4 and 5), inserts elements in lines 9–18, and deletes elements in lines 22–28. Since **AVLTree** is a subclass of **BST** and the elements in a **BST** are iterable, the program uses a foreach loop to traverse all the elements in lines 33–35.

Listing 26.4 TestAVLTree.java

```
 1 public class TestAVLTree {
 2   public static void main(String[] args) {
 3     // Create an AVL tree
 4     AVLTree<Integer> tree = new AVLTree<Integer>(new Integer[]{25,
 5       20, 5});
 6     System.out.print("After inserting 25, 20, 5:");
 7     printTree(tree);
 8
 9     tree.insert(34);
10     tree.insert(50);
11     System.out.print("\nAfter inserting 34, 50:");
12     printTree(tree);
13
14     tree.insert(30);
15     System.out.print("\nAfter inserting 30");
16     printTree(tree);
17
18     tree.insert(10);
19     System.out.print("\nAfter inserting 10");
20     printTree(tree);
21
22     tree.delete(34);
23     tree.delete(30);
24     tree.delete(50);
25     System.out.print("\nAfter removing 34, 30, 50:");
26     printTree(tree);
27
28     tree.delete(5);
29     System.out.print("\nAfter removing 5:");
30     printTree(tree);
31
32     System.out.print("\nTraverse the elements in the tree: ");
33     for (int e: tree) {
34       System.out.print(e + " ");
35     }
36   }
37
38   public static void printTree(BST tree) {
39     // Traverse tree
40     System.out.print("\nInorder (sorted): ");
41     tree.inorder();
42     System.out.print("\nPostorder: ");
43     tree.postorder();
44     System.out.print("\nPreorder: ");
45     tree.preorder();
46     System.out.print("\nThe number of nodes is " + tree.getSize());
```

create an AVLTree

insert 34

insert 50

insert 30

insert 10

delete 34

delete 30

delete 50

delete 5

foreach loop

```
    System.out.println();
  }
}
```

```
After inserting 25, 20, 5:
Inorder (sorted): 5 20 25
Postorder: 5 25 20
Preorder: 20 5 25
The number of nodes is 3

After inserting 34, 50:
Inorder (sorted): 5 20 25 34 50
Postorder: 5 25 50 34 20
Preorder: 20 5 34 25 50
The number of nodes is 5

After inserting 30
Inorder (sorted): 5 20 25 30 34 50
Postorder: 5 20 30 50 34 25
Preorder: 25 20 5 34 30 50
The number of nodes is 6

After inserting 10
Inorder (sorted): 5 10 20 25 30 34 50
Postorder: 5 20 10 30 50 34 25
Preorder: 25 10 5 20 34 30 50
The number of nodes is 7

After removing 34, 30, 50:
Inorder (sorted): 5 10 20 25
Postorder: 5 20 25 10
Preorder: 10 5 25 20
The number of nodes is 4

After removing 5:
Inorder (sorted): 10 20 25
Postorder: 10 25 20
Preorder: 20 10 25
The number of nodes is 3
Traverse the elements in the tree: 10 20 25
```

Figure 26.10 shows how the tree evolves as elements are added to the tree. After **25** and **20** are added, the tree is as shown in Figure 26.10a. **5** is inserted as a left child of **20**, as shown in Figure 26.10b. The tree is not balanced. It is left-heavy at node **25**. Perform an LL rotation to result in an AVL tree as shown in Figure 26.10c.

After inserting **34**, the tree is as shown in Figure 26.10d. After inserting **50**, the tree is as shown in Figure 26.10e. The tree is not balanced. It is right-heavy at node **25**. Perform an RR rotation to result in an AVL tree as shown in Figure 26.10f.

After inserting **30**, the tree is as shown in Figure 26.10g. The tree is not balanced. Perform an RL rotation to result in an AVL tree as shown in Figure 26.10h.

After inserting **10**, the tree is as shown in Figure 26.10i. The tree is not balanced. Perform an LR rotation to result in an AVL tree as shown in Figure 26.10j.

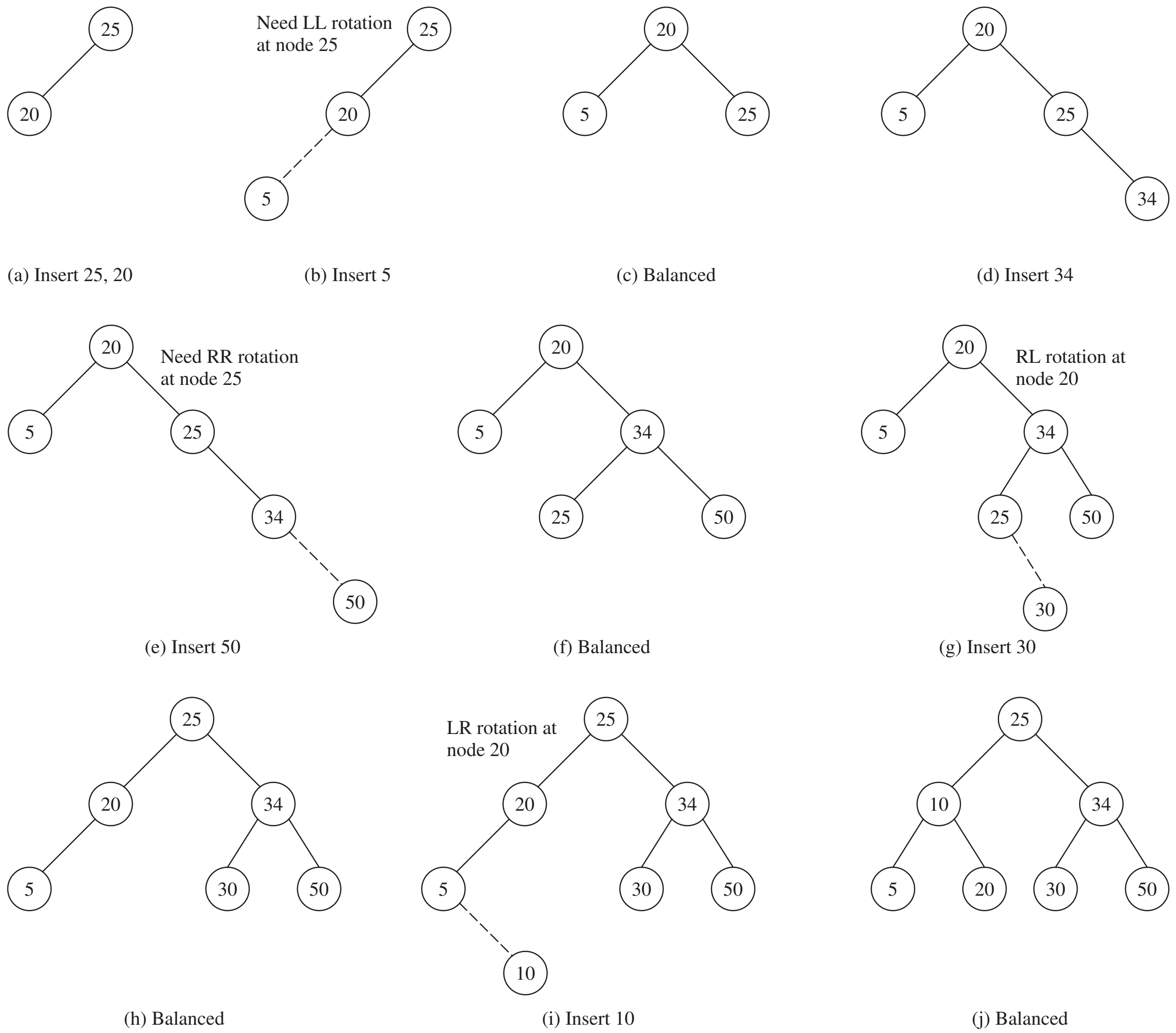

25
20
Need LL rotation at node 25
25
20
5
20
5
25
20
5
25
34
(a) Insert 25, 20
(b) Insert 5
(c) Balanced
(d) Insert 34
20
Need RR rotation at node 25
5
25
34
50
20
5
34
25
50
20
RL rotation at node 20
5
34
25
50
30
(e) Insert 50
(f) Balanced
(g) Insert 30
25
20
34
5
30
50
LR rotation at node 20
25
20
34
5
30
50
10
25
10
34
5
20
30
50
(h) Balanced
(i) Insert 10
(j) Balanced

FIGURE 26.10 The tree evolves as new elements are inserted.

Figure 26.11a shows how the tree evolves as elements are deleted. After deleting **34**, **30**, and **50**, the tree is as shown in Figure 26.11b. The tree is not balanced. Perform an LL rotation to result in an AVL tree as shown in Figure 26.11c.

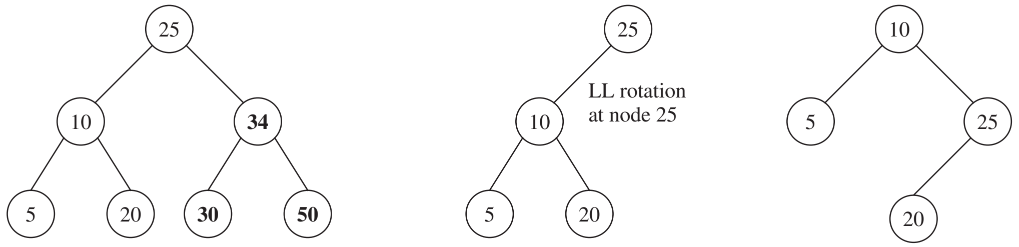
After deleting **5**, the tree is as shown in Figure 26.11d. The tree is not balanced. Perform an RL rotation to result in an AVL tree as shown in Figure 26.11e.

Check Point

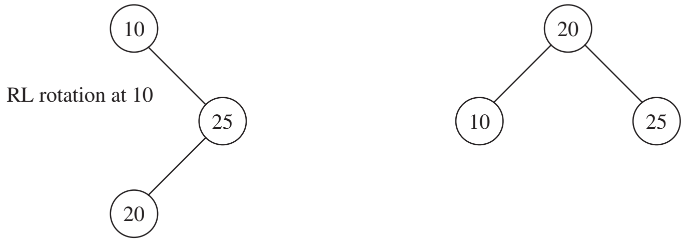
26.8.1 Show the change of an AVL tree when inserting **1**, **2**, **3**, **4**, **10**, **9**, **7**, **5**, **8**, **6** into the tree, in this order.

26.8.2 For the tree built in the preceding question, show its change after **1**, **2**, **3**, **4**, **10**, **9**, **7**, **5**, **8**, **6** are deleted from the tree in this order.

26.8.3 Can you traverse the elements in an AVL tree using a foreach loop?



(a) Delete 34, 30, 50 (b) After 34, 30, 50 are deleted (c) Balanced



(d) After 5 is deleted (e) Balanced

FIGURE 26.11 The tree evolves as elements are deleted from the tree.

26.9 AVL Tree Time Complexity Analysis

Key Point

Since the height of an AVL tree is O(log n), the time complexity of the `search`*,* `insert`*, and* `delete` *methods in* `AVLTree` *is O(log n).*

tree height

The time complexity of the `search`, `insert`, and `delete` methods in `AVLTree` depends on the height of the tree. We can prove that the height of the tree is $O(\log n)$.

Let $G(h)$ denote the minimum number of nodes in an AVL tree with height h. For the definition of the height of a binary tree, see Section 25.2. Obviously, $G(0)$ is 1 and $G(1)$ is 2. The minimum number of nodes in an AVL tree with height $h \geq 2$ must have two minimum subtrees: one with height $h - 1$ and the other with height $h - 2$. Thus,

$$G(h) = G(h - 1) + G(h - 2) + 1$$

Recall that a Fibonacci number at index i can be described using the recurrence relation $F(i) = F(i - 1) + F(i - 2)$. Therefore, the function $G(h)$ is essentially the same as $F(i)$. It can be proven that

$$h < 1.4405 \log(n + 2) - 1.3277$$

where n is the number of nodes in the tree. Hence, the height of an AVL tree is $O(\log n)$.

The `search`, `insert`, and `delete` methods involve only the nodes along a path in the tree. The `updateHeight` and `balanceFactor` methods are executed in a constant time for each node in the path. The `balancePath` method is executed in a constant time for a node in the path. Thus, the time complexity for the `search`, `insert`, and `delete` methods is $O(\log n)$.

Check Point

26.9.1 What is the maximum/minimum height for an AVL tree of 3 nodes, 5 nodes, and 7 nodes?

26.9.2 If an AVL tree has a height of 3, what maximum number of nodes can the tree have? What minimum number of nodes can the tree have?

26.9.3 If an AVL tree has a height of 4, what maximum number of nodes can the tree have? What minimum number of nodes can the tree have?

KEY TERMS

AVL tree	996
balance factor	996
left-heavy	996
LL rotation	996
LR rotation	997
perfectly balanced tree	996
right-heavy	996
RL rotation	997
rotation	996
RR rotation	996
well-balanced tree	996

CHAPTER SUMMARY

1. An *AVL tree* is a *well-balanced* binary tree. In an AVL tree, the difference between the heights of two subtrees for every node is `0` or `1`.

2. The process for inserting or deleting an element in an AVL tree is the same as in a binary search tree. The difference is that you may have to rebalance the tree after an insertion or deletion operation.

3. Imbalances in the tree caused by insertions and deletions are rebalanced through subtree rotations at the node of the imbalance.

4. The process of rebalancing a node is called a *rotation*. There are four possible rotations: *LL rotation*, *LR rotation*, *RR rotation*, and *RL rotation*.

5. The height of an AVL tree is $O(\log n)$. Therefore, the time complexities for the `search`, `insert`, and `delete` methods are $O(\log n)$.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

***26.1** (*Display AVL tree graphically*) Write a program that displays an AVL tree along with its balance factor for each node.

26.2 (*Compare performance*) Write a test program that randomly generates 500,000 numbers and inserts them into a `BST`, reshuffles the 500,000 numbers and performs a search, and reshuffles the numbers again before deleting them from the tree. Write another test program that does the same thing for an `AVLTree`. Compare the execution times of these two programs.

*****26.3** (*AVL tree animation*) Write a program that animates the AVL tree `insert`, `delete`, and `search` methods, as shown in Figure 26.2.

****26.4** (*Parent reference for `BST`*) Suppose the `TreeNode` class defined in `BST` contains a reference to the node's parent, as shown in Programming Exercise 25.15. Implement the `AVLTree` class to support this change. Write a test program that adds numbers `1`, `2`, . . . , `100` to the tree and displays the paths for all leaf nodes.

****26.5** (*The kth smallest element*) You can find the kth smallest element in a BST in $O(n)$ time from an inorder iterator. For an AVL tree, you can find it in $O(\log n)$ time. To achieve this, add a new data field named `size` in `AVLTreeNode` to store the number of nodes in the subtree rooted at this node. Note the size of a node v is one more than the sum of the sizes of its two children. Figure 26.12 shows an AVL tree and the `size` value for each node in the tree.

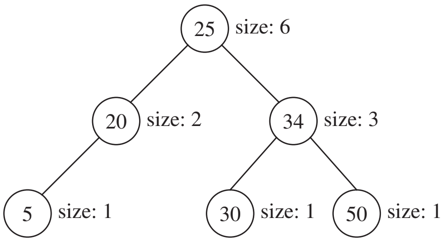


FIGURE 26.12 The `size` data field in `AVLTreeNode` stores the number of nodes in the subtree rooted at the node.

In the `AVLTree` class, add the following method to return the *k*th smallest element in the tree:

```
public E find(int k)
```

The method returns `null` if `k < 1` or `k > the size of the tree`. This method can be implemented using the recursive method `find(k, root)`, which returns the *k*th smallest element in the tree with the specified root. Let `A` and `B` be the left and right children of the root, respectively. Assuming the tree is not empty and $k \leq root.size$, `find(k, root)` can be recursively defined as follows:

$$find(k, root) = \begin{cases} root.element, \text{ if } A \text{ is null and } k \text{ is } 1; \\ B.element, \text{ if } A \text{ is null and } k \text{ is } 2; \\ find(k, A), \text{ if } k <= A.size; \\ root.element, \text{ if } k = A.size + 1; \\ find(k - A.size - 1, B), \text{ if } k > A.size + 1; \end{cases}$$

Modify the `insert` and `delete` methods in `AVLTree` to set the correct value for the `size` property in each node. The `insert` and `delete` methods will still be in $O(\log n)$ time. The `find(k)` method can be implemented in $O(\log n)$ time. Therefore, you can find the *k*th smallest element in an AVL tree in $O(\log n)$ time.

Test your program using the code at https://liveexample.pearsoncmg.com/test/Exercise26_05.txt.

****26.6** (*Closest pair of points*) Section 22.8 introduced an algorithm for finding a closest pair of points in O(nlogn) time using a divide-and-conquer approach. The algorithm was implemented using recursion with a lot of overhead. Using an AVL tree, you can solve the same problem in O(nlogn) time. Implement the algorithm using an `AVLTree.`

****26.7** (*Test AVL tree*) Define a new class named `MyBST` that extends the `BST` class with the following method:

```
// Returns true if the tree is an AVL tree
public boolean isAVLTree()
```

Use https://liveexample.pearsoncmg.com/test/Exercise26_07.txt to test your code.

CHAPTER 27

HASHING

Objectives

- To understand what hashing is and for what hashing is used (§27.2).
- To obtain the hash code for an object and design the hash function to map a key to an index (§27.3).
- To handle collisions using open addressing (§27.4).
- To know the differences among linear probing, quadratic probing, and double hashing (§27.4).
- To handle collisions using separate chaining (§27.5).
- To understand the load factor and the need for rehashing (§27.6).
- To implement `MyHashMap` using hashing (§27.7).
- To implement `MyHashSet` using hashing (§27.8).

27.1 Introduction

Key Point

Hashing is superefficient. It takes O(1) time to search, insert, and delete an element using hashing.

why hashing?

The preceding chapter introduced binary search trees. An element can be found in $O(\log n)$ time in a well-balanced search tree. Is there a more efficient way to search for an element in a container? This chapter introduces a technique called *hashing*. You can use hashing to implement a map or a set to search, insert, and delete an element in $O(1)$ time.

27.2 What Is Hashing?

Key Point

Hashing uses a hashing function to map a key to an index.

map
key
value

Before introducing hashing, let us review map, which is a data structure that is implemented using hashing. Recall that a *map* (introduced in Section 21.5) is a container object that stores entries. Each entry contains two parts: a *key* and a *value*. The key, also called a *search key*, is used to search for the corresponding value. For example, a dictionary can be stored in a map, in which the words are the keys and the definitions of the words are the values.

dictionary

Note

A map is also called a *dictionary*, a *hash table*, or an *associative array*.

associative array

The Java Collections Framework defines the `java.util.Map` interface for modeling maps. Three concrete implementations are `java.util.HashMap`, `java.util.LinkedHashMap`, and `java.util.TreeMap`. `java.util.HashMap` is implemented using hashing, `java.util.LinkedHashMap` using `LinkedList`, and `java.util.TreeMap` using red–black trees. (Bonus Chapter 41 will introduce red–black trees.) You will learn the concept of hashing and use it to implement a hash map in this chapter.

hash table
hash function
hashing

If you know the index of an element in the array, you can retrieve the element using the index in $O(1)$ time. So does that mean we can store the values in an array and use the key as the index to find the value? The answer is yes—if you can map a key to an index. The array that stores the values is called a *hash table*. The function that maps a key to an index in the hash table is called a *hash function*. As shown in Figure 27.1, a hash function obtains an index from a key and uses the index to retrieve the value for the key. *Hashing* is a technique that retrieves the value using the index obtained from the key without performing a search.

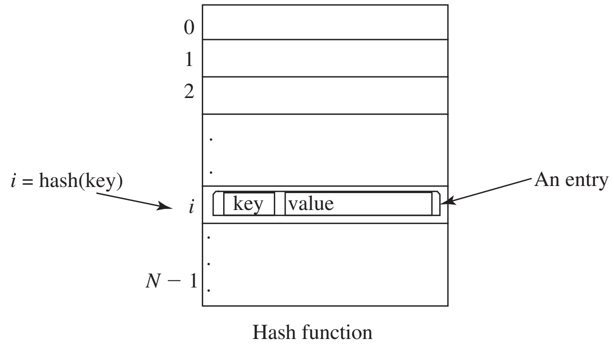


FIGURE 27.1 A hash function maps a key to an index in the hash table.

perfect hash function

How do you design a hash function that produces an index from a key? Ideally, we would like to design a function that maps each search key to a different index in the hash table. Such a function is called a *perfect hash function*. However, it is difficult to find a perfect hash function. When two or more keys are mapped to the same hash value, we say a *collision* has occurred. Although there are ways to deal with collisions, which will be are discussed later in this chapter, it is better to avoid collisions in the first place. Thus, you should design a fast and easy-to-compute hash function that minimizes collisions.

collision

Check Point

27.2.1 What is a hash function? What is a perfect hash function? What is a collision?

27.3 Hash Functions and Hash Codes

Key Point

A typical hash function first converts a search key to an integer value called a hash code, then compresses the hash code into an index to the hash table.

A hash code is a number generated from an object. This code allows an object to be stored/retrieved quickly in a hash table. Java's root class `Object` has the `hashCode()` method, which returns an integer *hash code*. By default, the method returns the memory address for the object. The general contract for the `hashCode` method is as follows:

hash code

`hashCode()`

1. You should override the `hashCode` method whenever the `equals` method is overridden to ensure two equal objects return the same hash code.
2. During the execution of a program, invoking the `hashCode` method multiple times returns the same integer, provided that the object's data are not changed.
3. Two unequal objects may have the same hash code, but you should implement the `hashCode` method to avoid too many such cases.

27.3.1 Hash Codes for Primitive Types

For search keys of the type `byte`, `short`, `int`, and `char`, simply cast them to `int`. Therefore, two different search keys of any one of these types will have different hash codes.

`byte, short, int, char`

For a search key of the type `float`, use `Float.floatToIntBits(key)` as the hash code. Note `floatToIntBits(float f)` returns an `int` value whose bit representation is the same as the bit representation for the floating number `f`. Thus, two different search keys of the `float` type will have different hash codes.

`float`

For a search key of the type `long`, simply casting it to `int` would not be a good choice, because all keys that differ in only the first 32 bits will have the same hash code. To take the first 32 bits into consideration, divide the 64 bits into two halves and perform the exclusive-or operation to combine the two halves. This process is called *folding*. The hash code for a `long` key is

`long`

folding

```
int hashCode = (int)(key ^ (key >> 32));
```

Note `>>` is the right-shift operator that shifts the bits 32 positions to the right. For example, `1010110 >> 2` yields `0010101`. The `^` is the bitwise exclusive-or operator. It operates on two corresponding bits of the binary operands. For example, `1010110 ^ 0110111` yields `1100001`. For more on bitwise operations, see Appendix G, Bitwise Operations.

For a search key of the type `double`, first convert it to a `long` value using the `Double.doubleToLongBits` method, then perform a folding as follows:

`double`

folding

```
long bits = Double.doubleToLongBits(key);
int hashCode = (int)(bits ^ (bits >> 32));
```

27.3.2 Hash Codes for Strings

Search keys are often strings, so it is important to design a good hash function for strings. An intuitive approach is to sum the Unicode of all characters as the hash code for the string. This approach may work if two search keys in an application don't contain the same letters, but it will produce a lot of collisions if the search keys contain the same letters, such as `tod` and `dot`.

A better approach is to generate a hash code that takes the position of characters into consideration. Specifically, let the hash code be

$$s_0 * b^{(n-1)} + s_1 * b^{(n-2)} + \cdots + s_{n-1}$$

polynomial hash code

where s_i is `s.charAt(i)`. This expression is a polynomial for some positive b, so this is called a *polynomial hash code*. Using Horner's rule for polynomial evaluation (see Section 6.7), the hash code can be calculated efficiently as follows:

$$(\cdots((s_0 * b + s_1) * b + s_2) * b + \cdots + s_{n-2}) * b + s_{n-1}$$

This computation can cause an overflow for long strings, but arithmetic overflow is ignored in Java. You should choose an appropriate value b to minimize collisions. Experiments show that good choices for b are 31, 33, 37, 39, and 41. In the `String` class, the `hashCode` is overridden using the polynomial hash code with b being `31`.

27.3.3 Compressing Hash Codes

The hash code for a key can be a large integer that is out of the range for the hash-table index, so you need to scale it down to fit in the index's range. Assume the index for a hash table is between `0` and `N-1`. The most common way to scale an integer to a number between `0` and `N-1` is to use

```
index = hashCode % N;
```

Ideally, you should choose a prime number for `N` to ensure the indices are spread evenly. However, it is time consuming to find a large prime number. In the Java API implementation for `java.util.HashMap`, `N` is set to an integer power of `2`. There is a good reason for this choice. When `N` is an `int` value power of `2`, you can use the `&` operator to compress a hash code to an index on the hash table as follows:

```
index = hashCode & (N – 1);
```

`index` will be between `0` and `N – 1`. The ampersand, `&`, is a bitwise AND operator (see Appendix G, Bitwise Operations). The AND of two corresponding bits yields a `1` if both bits are `1`. For example, assume `N = 4` and `hashCode = 11`. Thus, `11 & (4 – 1) = 1011 & 0011 = 0011`.

To ensure the hashing is evenly distributed, a supplemental hash function is also used along with the primary hash function in the implementation of `java.util.HashMap`. This function is defined as

```
private static int supplementalHash(int h) {
  h ^= (h >>> 20) ^ (h >>> 12);
  return h ^ (h >>> 7) ^ (h >>> 4);
}
```

`^` and `>>>` are bitwise exclusive-or and unsigned right-shift operations (see Appendix G). The bitwise operations are much faster than the multiplication, division, and remainder operations. You should replace these operations with the bitwise operations whenever possible.

The complete hash function is defined as

```
h(hashCode) = supplementalHash(hashCode) & (N – 1)
```

The supplemental hash function helps avoid collisions for two numbers with the same lower bits. For example, both `11100101 & 00000111` and `11001101 & 00000111` yield `00000111`. But `supplementalHash(11100101) & 00000111` and `supplementalHash(11001101) & 00000111` will be different. Using a supplemental function reduces this type of collision.

Note

In Java, an `int` is a 32-bit signed integer. The `hashCode()` method returns an `int` and it may be negative. If a hash code is negative, `hashCode % N` would be negative. But `hashCode & (N – 1)` will be non-negative for an `int` value `N` because `anyInt & aNonNegativeInt` is always non-negative.

Check Point

27.3.1 What is a hash code? What is the hash code for `Byte`, `Short`, `Integer`, and `Character`?

27.3.2 How is the hash code for a `Float` object computed?

27.3.3 How is the hash code for a `Long` object computed?

27.3.4 How is the hash code for a `Double` object computed?

27.3.5 How is the hash code for a `String` object computed?

27.3.6 How is a hash code compressed to an integer representing the index in a hash table?

27.3.7 If `N` is an integer power of the power of `2`, is `N / 2` same as `N >> 1`?

27.3.8 If `N` is an integer power of the power of `2`, is `m % N` same as `m & (N – 1)` for a positive integer `m`?

27.3.9 What is `Integer.valueOf("-98").hashCode()` and what is `"ABCDEFGHIJK".hashCode()`?

27.4 Handling Collisions Using Open Addressing

Key Point

A collision occurs when two keys are mapped to the same index in a hash table. Generally, there are two ways for handling collisions: open addressing and separate chaining.

Open addressing is the process of finding an open location in the hash table in the event of a collision. Open addressing has several variations: *linear probing*, *quadratic probing*, and *double hashing*.

open addressing

27.4.1 Linear Probing

When a collision occurs during the insertion of an entry to a hash table, *linear probing* finds the next available location sequentially. For example, if a collision occurs at `hashTable[k % N]`, check whether `hashTable[(k+1) % N]` is available. If not, check `hashTable[(k+2) % N]` and so on, until an available cell is found, as shown in Figure 27.2.

add entry

linear probing

Note

When probing reaches the end of the table, it goes back to the beginning of the table. Thus, the hash table is treated as if it were circular.

circular hash table

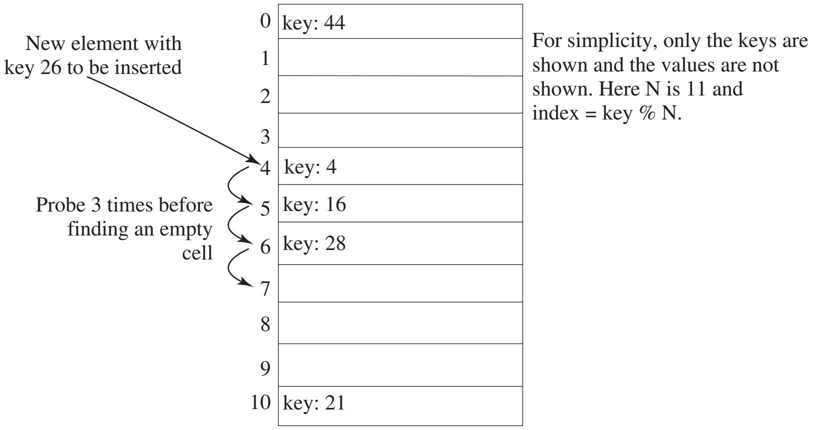

New element with key 26 to be inserted
Probe 3 times before finding an empty cell
0
1
2
3
4
5
6
7
8
9
10
key: 44
key: 4
key: 16
key: 28
key: 21
For simplicity, only the keys are shown and the values are not shown. Here N is 11 and index = key % N.

FIGURE 27.2 Linear probing finds the next available location sequentially.

search entry

To search for an entry in the hash table, obtain the index, say `k`, from the hash function for the key. Check whether `hashTable[k % N]` contains the entry. If not, check whether `hashTable[(k+1) % N]` contains the entry, and so on, until it is found, or an empty cell is reached.

remove entry

To remove an entry from the hash table, search the entry that matches the key. If the entry is found, place a special marker to denote that the entry is available. Each cell in the hash table has three possible states: occupied, marked, or empty. Note a marked cell is also available for insertion.

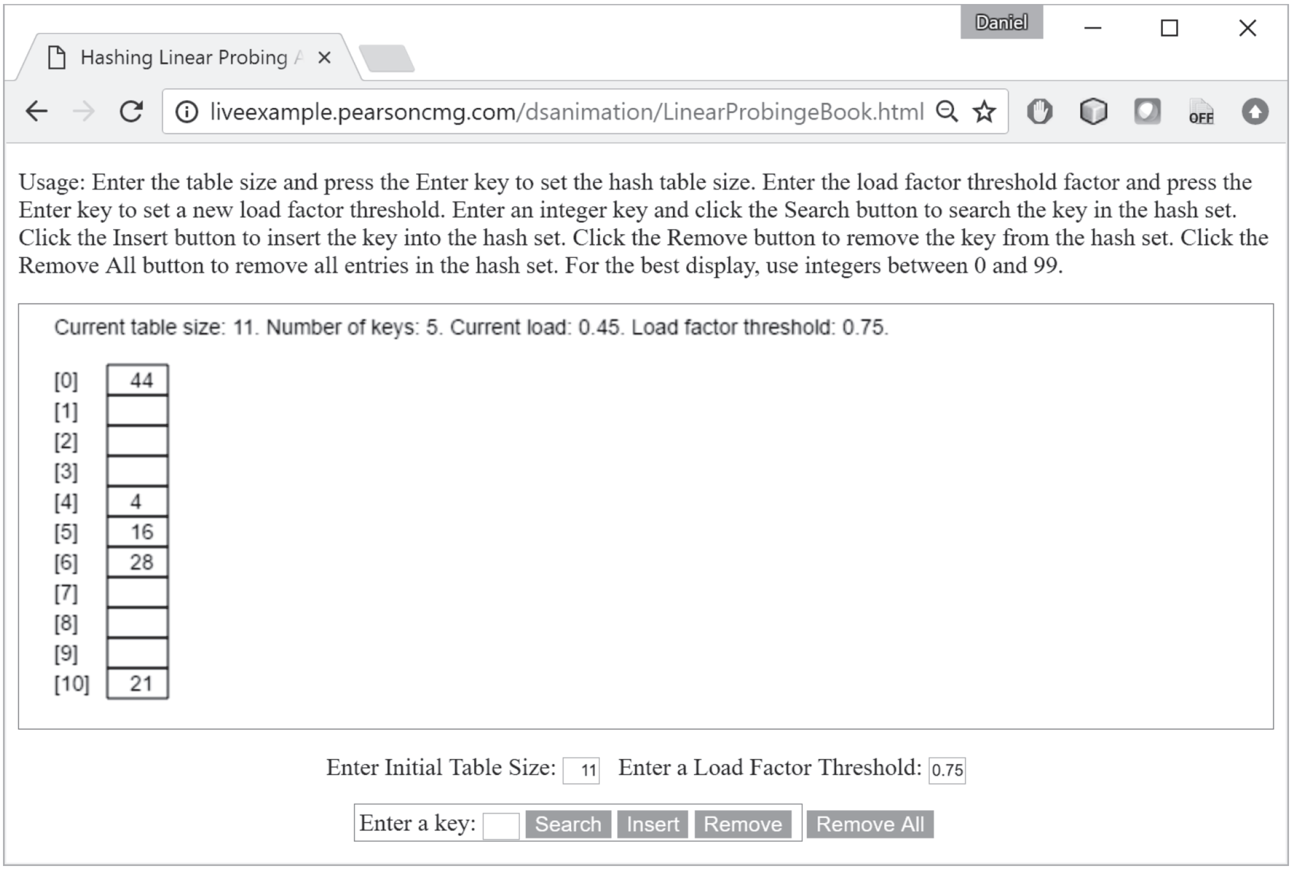
cluster

Linear probing tends to cause groups of consecutive cells in the hash table to be occupied. Each group is called a *cluster*. Each cluster is actually a probe sequence that you must search when retrieving, adding, or removing an entry. As clusters grow in size, they may merge into even larger clusters, further slowing down the search time. This is a big disadvantage of linear probing.

linear probing animation on Companion Website

Pedagogical Note

For an interactive GUI demo to see how linear probing works, go to http://liveexample.pearsoncmg.com/dsanimation/LinearProbingeBook.html, as shown in Figure 27.3.


Daniel
Hashing Linear Probing
liveexample.pearsoncmg.com/dsanimation/LinearProbingeBook.html
OFF

Usage: Enter the table size and press the Enter key to set the hash table size. Enter the load factor threshold factor and press the Enter key to set a new load factor threshold. Enter an integer key and click the Search button to search the key in the hash set. Click the Insert button to insert the key into the hash set. Click the Remove button to remove the key from the hash set. Click the Remove All button to remove all entries in the hash set. For the best display, use integers between 0 and 99.

Current table size: 11. Number of keys: 5. Current load: 0.45. Load factor threshold: 0.75.

[0]	44
[1]	
[2]	
[3]	
[4]	4
[5]	16
[6]	28
[7]	
[8]	
[9]	
[10]	21

Enter Initial Table Size: 11 Enter a Load Factor Threshold: 0.75

Enter a key:

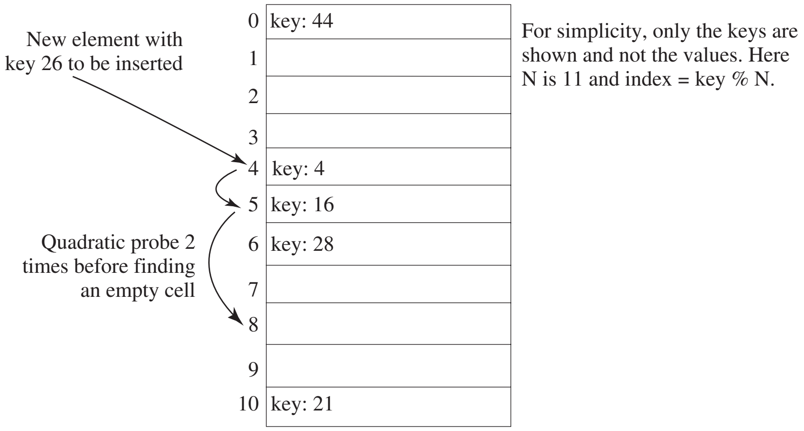
FIGURE 27.3 The animation tool shows how linear probing works.

27.4.2 Quadratic Probing

quadratic probing

Quadratic probing can avoid the clustering problem that can occur in linear probing. Linear probing looks at the consecutive cells beginning at index k. Quadratic probing, on the other hand, looks at the cells at indices $(k + j^2) \% N$, for $j \geq 0$, that is, $k \% N$, $(k + 1) \% N$, $(k + 4) \% n$, $(k + 9) \% N$, and so on, as shown in Figure 27.4.

Quadratic probing works in the same way as linear probing except for a change in the search sequence. Quadratic probing avoids linear probing's clustering problem, but it has its own


New element with key 26 to be inserted
Quadratic probe 2 times before finding an empty cell
0 key: 44
1
2
3
4 key: 4
5 key: 16
6 key: 28
7
8
9
10 key: 21

For simplicity, only the keys are shown and not the values. Here N is 11 and index = key % N.

FIGURE 27.4 Quadratic probing increases the next index in the sequence by j^2 for $j = 1, 2, 3, \ldots$.

clustering problem, called *secondary clustering*; that is, the entries that collide with an occupied entry use the same probe sequence.

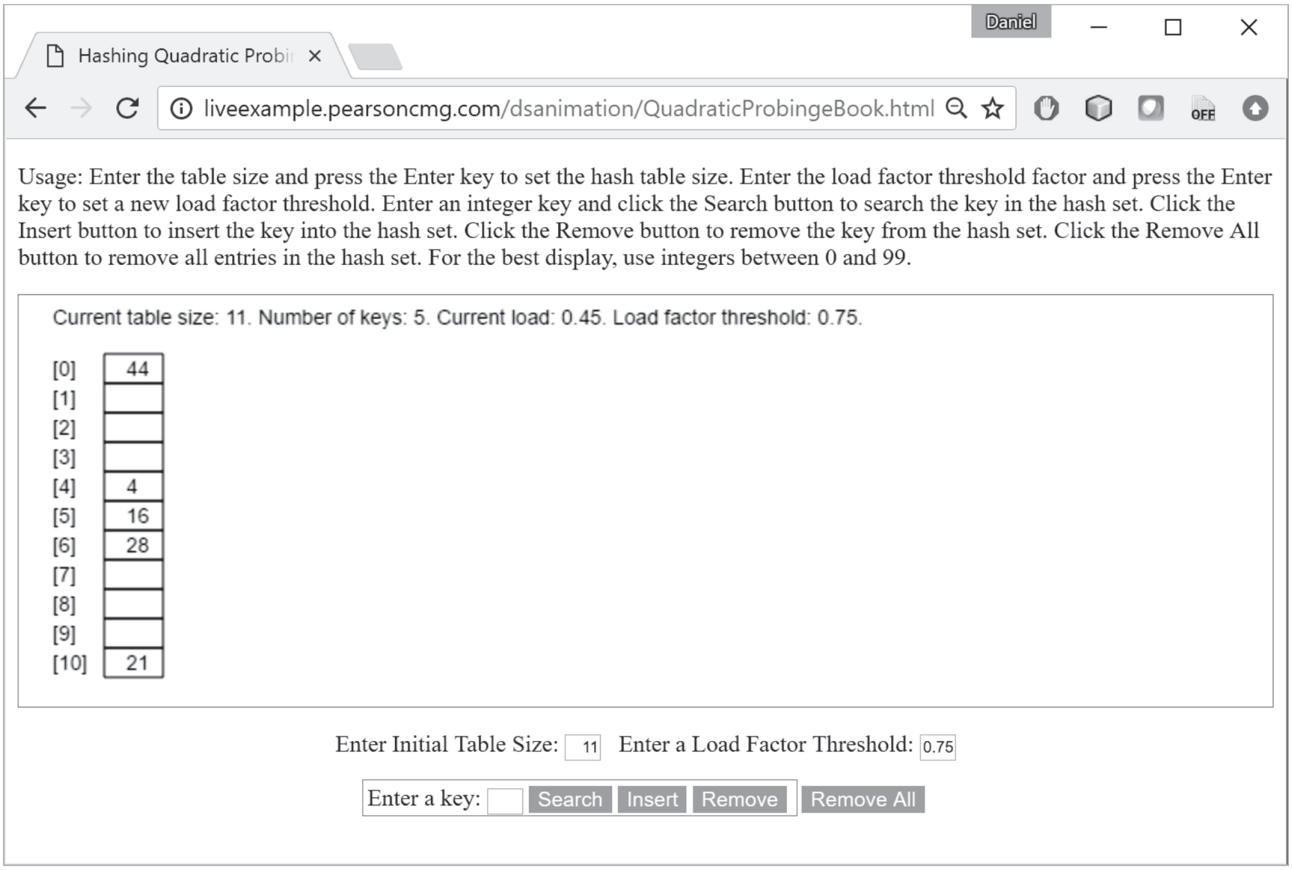
secondary clustering

Linear probing guarantees that an available cell can be found for insertion as long as the table is not full. However, there is no such guarantee for quadratic probing.

Pedagogical Note

For an interactive GUI demo to see how quadratic probing works, go to http://liveexample.pearsoncmg.com/dsanimation/QuadraticProbingeBook.html, as shown in Figure 27.5.

quadratic probing animation on Companion Website


Daniel
Hashing Quadratic Probi
liveexample.pearsoncmg.com/dsanimation/QuadraticProbingeBook.html
OFF
Search
Insert
Remove
Remove All

Usage: Enter the table size and press the Enter key to set the hash table size. Enter the load factor threshold factor and press the Enter key to set a new load factor threshold. Enter an integer key and click the Search button to search the key in the hash set. Click the Insert button to insert the key into the hash set. Click the Remove button to remove the key from the hash set. Click the Remove All button to remove all entries in the hash set. For the best display, use integers between 0 and 99.

Current table size: 11. Number of keys: 5. Current load: 0.45. Load factor threshold: 0.75.

[0]	44
[1]	
[2]	
[3]	
[4]	4
[5]	16
[6]	28
[7]	
[8]	
[9]	
[10]	21

Enter Initial Table Size: 11 Enter a Load Factor Threshold: 0.75

Enter a key:

FIGURE 27.5 The animation tool shows how quadratic probing works.

27.4.3 Double Hashing

double hashing

Another open addressing scheme that avoids the clustering problem is known as *double hashing*. Starting from the initial index *k*, both linear probing and quadratic probing add an increment to *k* to define a search sequence. The increment is `1` for linear probing and j^2 for quadratic probing. These increments are independent of the keys. Double hashing uses a secondary hash function $h'(key)$ on the keys to determine the increments to avoid the clustering problem. Specifically, double hashing looks at the cells at indices $(k + j * h'(key)) \% N$, for $j \geq 0$, that is, $k \% N$, $(k + h'(key)) \% N$, $(k + 2 * h'(key)) \% N$, $(k + 3 * h'(key)) \% N$, and so on.

For example, let the primary hash function `h` and secondary hash function `h'` on a hash table of size `11` be defined as follows:

```
h(key) = key %  11;
h'(key) = 7 – key % 7;
```

For a search key of `12`, we have

```
h(12) = 12 % 11 = 1;
h'(12) = 7 – 12 % 7 = 2;
```

Suppose the elements with the keys `45`, `58`, `4`, `28`, and `21` are already placed in the hash table as shown in Figure 27.6. We now insert the element with key `12`. The probe sequence for key `12` starts at index `1`. Since the cell at index `1` is already occupied, search the next cell at index `3 (1 +1 * 2)`. Since the cell at index `3` is already occupied, search the next cell at index `5 (1 + 2 * 2)`. Since the cell at index `5` is empty, the element for key `12` is now inserted at this cell.

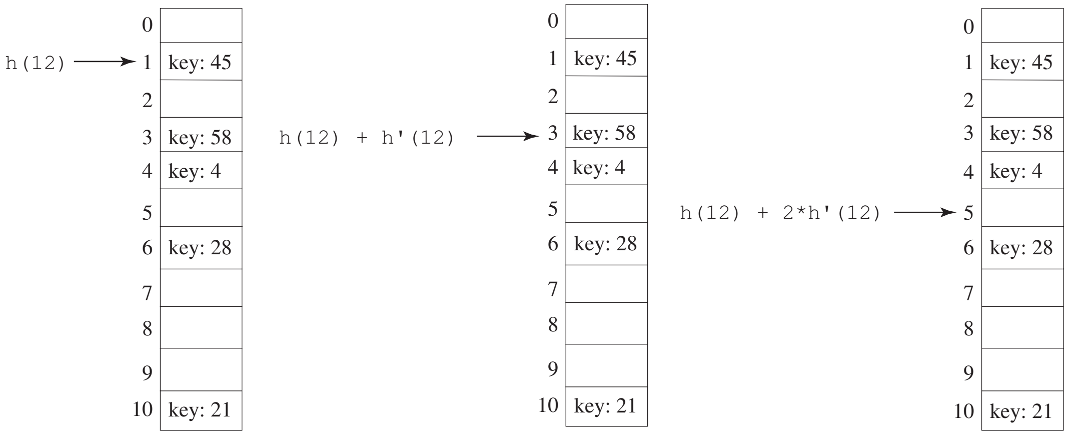


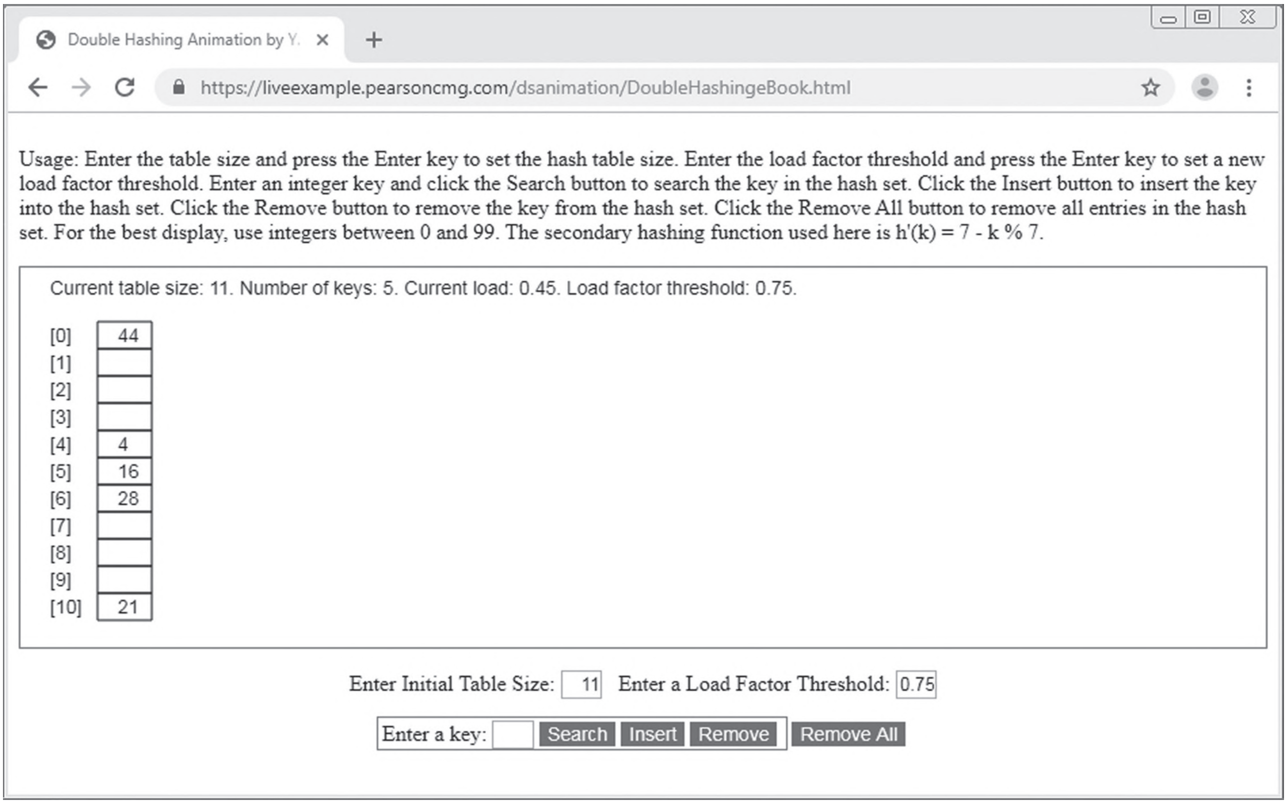
Figure 27.6 The secondary hash function in a double hashing determines the increment of the next index in the probe sequence.

The indices of the probe sequence are as follows: 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10. This sequence reaches the entire table. You should design your functions to produce a probe sequence that reaches the entire table. Note the second function should never have a zero value, since zero is not an increment.

double hashing animation on Companion Website

Pedagogical Note

For an interactive GUI demo to see how double hashing works, go to liveexample.pearsoncmg.com/dsanimation/DoubleHashingeBook.html, as shown in Figure 27.7.


Double Hashing Animation by Y.
https://liveexample.pearsoncmg.com/dsanimation/DoubleHashingeBook.html
Search
Insert
Remove
Remove All

Usage: Enter the table size and press the Enter key to set the hash table size. Enter the load factor threshold and press the Enter key to set a new load factor threshold. Enter an integer key and click the Search button to search the key in the hash set. Click the Insert button to insert the key into the hash set. Click the Remove button to remove the key from the hash set. Click the Remove All button to remove all entries in the hash set. For the best display, use integers between 0 and 99. The secondary hashing function used here is h'(k) = 7 - k % 7.

Current table size: 11. Number of keys: 5. Current load: 0.45. Load factor threshold: 0.75.

[0]	44
[1]	
[2]	
[3]	
[4]	4
[5]	16
[6]	28
[7]	
[8]	
[9]	
[10]	21

Enter Initial Table Size: 11 Enter a Load Factor Threshold: 0.75

Enter a key:

FIGURE 27.7 The animation tool shows how double hashing works.

Check Point

27.4.1 What is open addressing? What is linear probing? What is quadratic probing? What is double hashing?

27.4.2 Describe the clustering problem for linear probing.

27.4.3 What is secondary clustering?

27.4.4 Show the hash table of size 11 after inserting entries with keys 34, 29, 53, 44, 120, 39, 45, and 40, using linear probing.

27.4.5 Show the hash table of size 11 after inserting entries with keys 34, 29, 53, 44, 120, 39, 45, and 40, using quadratic probing.

27.4.6 Show the hash table of size 11 after inserting entries with keys 34, 29, 53, 44, 120, 39, 45, and 40, using double hashing with the following functions:

```
h(k) = k %  11;
h'(k) = 7 – k % 7;
```

27.5 Handling Collisions Using Separate Chaining

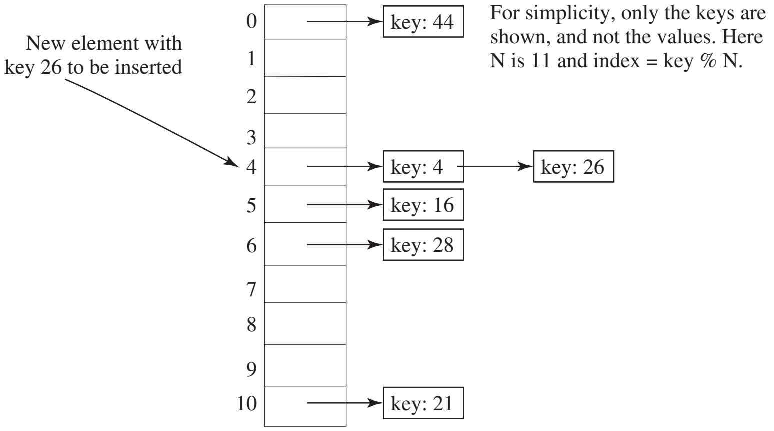
Key Point

The separate chaining scheme places all entries with the same hash index in the same location, rather than finding new locations. Each location in the separate chaining scheme uses a bucket to hold multiple entries.

separate chaining
implementing bucket

The preceding section introduced handling collisions using open addressing. The open addressing scheme finds a new location when a collision occurs. This section introduces handling collisions using separate chaining. The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a *bucket*. A bucket is a container that holds multiple entries.

You can implement a bucket using an array, `ArrayList`, or `LinkedList`. We will use `LinkedList` for demonstration. You can view each cell in the hash table as the reference to the head of a linked list, and elements in the linked list are chained starting from the head, as shown in Figure 27.8.


0
1
2
3
4
5
6
7
8
9
10
key: 44
key: 4
key: 26
key: 16
key: 28
key: 21

New element with key 26 to be inserted

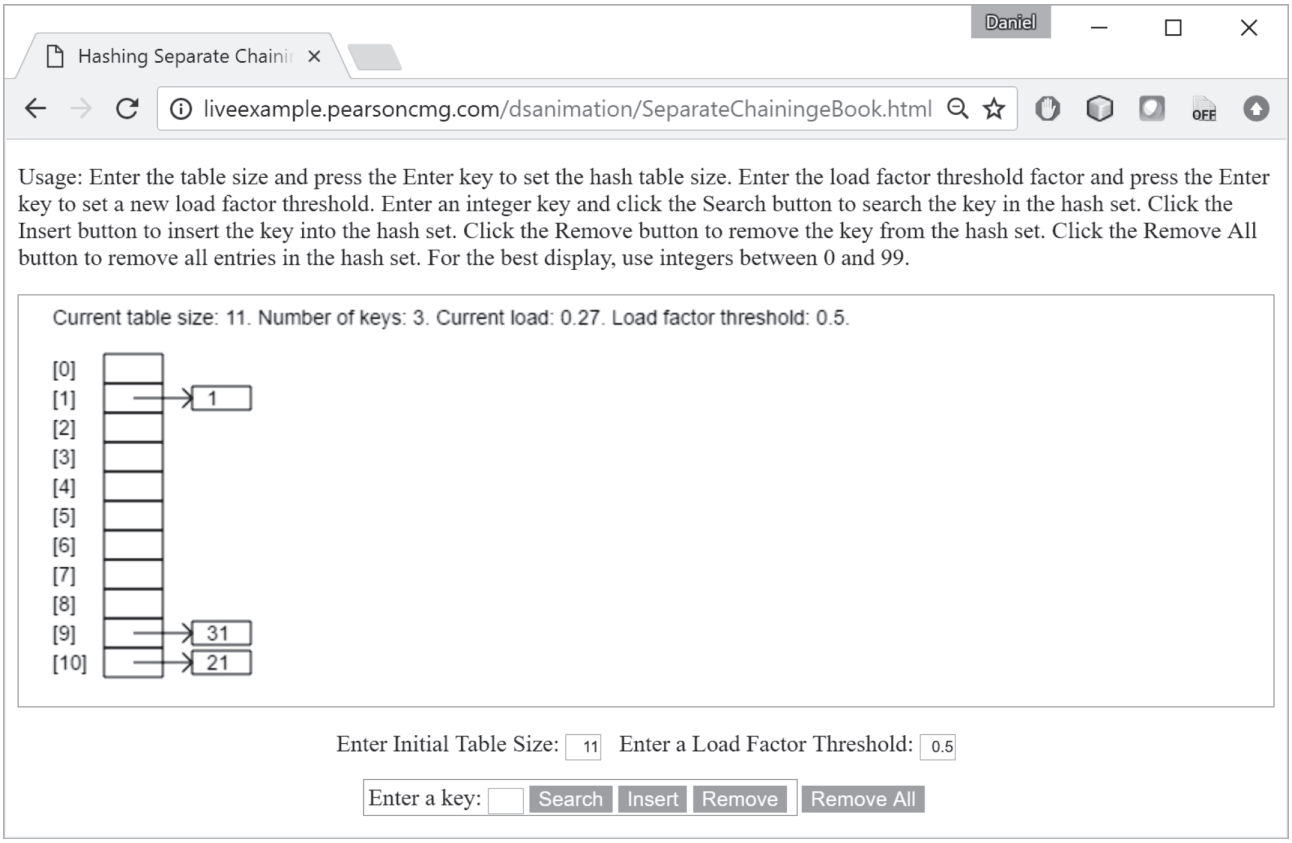
For simplicity, only the keys are shown, and not the values. Here N is 11 and index = key % N.

FIGURE 27.8 Separate chaining scheme chains the entries with the same hash index in a bucket.

separate chaining animation on Companion Website

Pedagogical Note

For an interactive GUI demo to see how separate chaining works, go to liveexample.pearsoncmg.com/dsanimation/SeparateChainingeBook.html, as shown in Figure 27.9.


Daniel
Hashing Separate Chaini
liveexample.pearsoncmg.com/dsanimation/SeparateChainingeBook.html
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
1
31
21
Search
Insert
Remove
Remove All

Usage: Enter the table size and press the Enter key to set the hash table size. Enter the load factor threshold factor and press the Enter key to set a new load factor threshold. Enter an integer key and click the Search button to search the key in the hash set. Click the Insert button to insert the key into the hash set. Click the Remove button to remove the key from the hash set. Click the Remove All button to remove all entries in the hash set. For the best display, use integers between 0 and 99.

Current table size: 11. Number of keys: 3. Current load: 0.27. Load factor threshold: 0.5.

Enter Initial Table Size: 11 Enter a Load Factor Threshold: 0.5

Enter a key:

FIGURE 27.9 The animation tool shows how separate chaining works.

27.5.1 Show the hash table of size 11 after inserting entries with the keys 34, 29, 53, 44, 120, 39, 45, and 40, using separate chaining.

Check Point

27.6 Load Factor and Rehashing

Key Point

The load factor measures how full a hash table is. If the load factor is exceeded, increase the hash-table size and reload the entries into a new larger hash table. This is called rehashing.

rehashing

Load factor

Load factor λ (*lambda*) measures how full a hash table is. It is the ratio of the number of elements to the size of the hash table, that is, $\lambda = \frac{n}{N}$, where n denotes the number of elements and N the size of the hash table.

Note λ is zero if the hash table is empty. For the open addressing scheme, λ is between `0` and `1`; λ is `1` if the hash table is full. For the separate chaining scheme, λ can be any value. As λ increases, the probability of a collision also increases. Studies show you should maintain the load factor under `0.5` for the open addressing scheme and under `0.9` for the separate chaining scheme.

threshold

rehash

Keeping the load factor under a certain threshold is important for the performance of hashing. In the implementation of the `java.util.HashMap` class in the Java API, the threshold `0.75` is used. Whenever the load factor exceeds the threshold, you need to increase the hash-table size and *rehash* all the entries in the map into a new larger hash table. Notice you need to change the hash functions, since the hash-table size has been changed. To reduce the likelihood of rehashing, since it is costly, you should at least double the hash-table size. Even with periodic rehashing, hashing is an efficient implementation for map.

Check Point

27.6.1 What is load factor? Assume the hash table has the initial size 4 and its load factor is 0.5; show the hash table after inserting entries with the keys 34, 29, 53, 44, 120, 39, 45, and 40, using linear probing.

27.6.2 Assume the hash table has the initial size 4 and its load factor is 0.5; show the hash table after inserting entries with the keys 34, 29, 53, 44, 120, 39, 45, and 40, using quadratic probing.

27.6.3 Assume the hash table has the initial size 4 and its load factor is 0.5; show the hash table after inserting entries with the keys 34, 29, 53, 44, 120, 39, 45, and 40, using separate chaining.

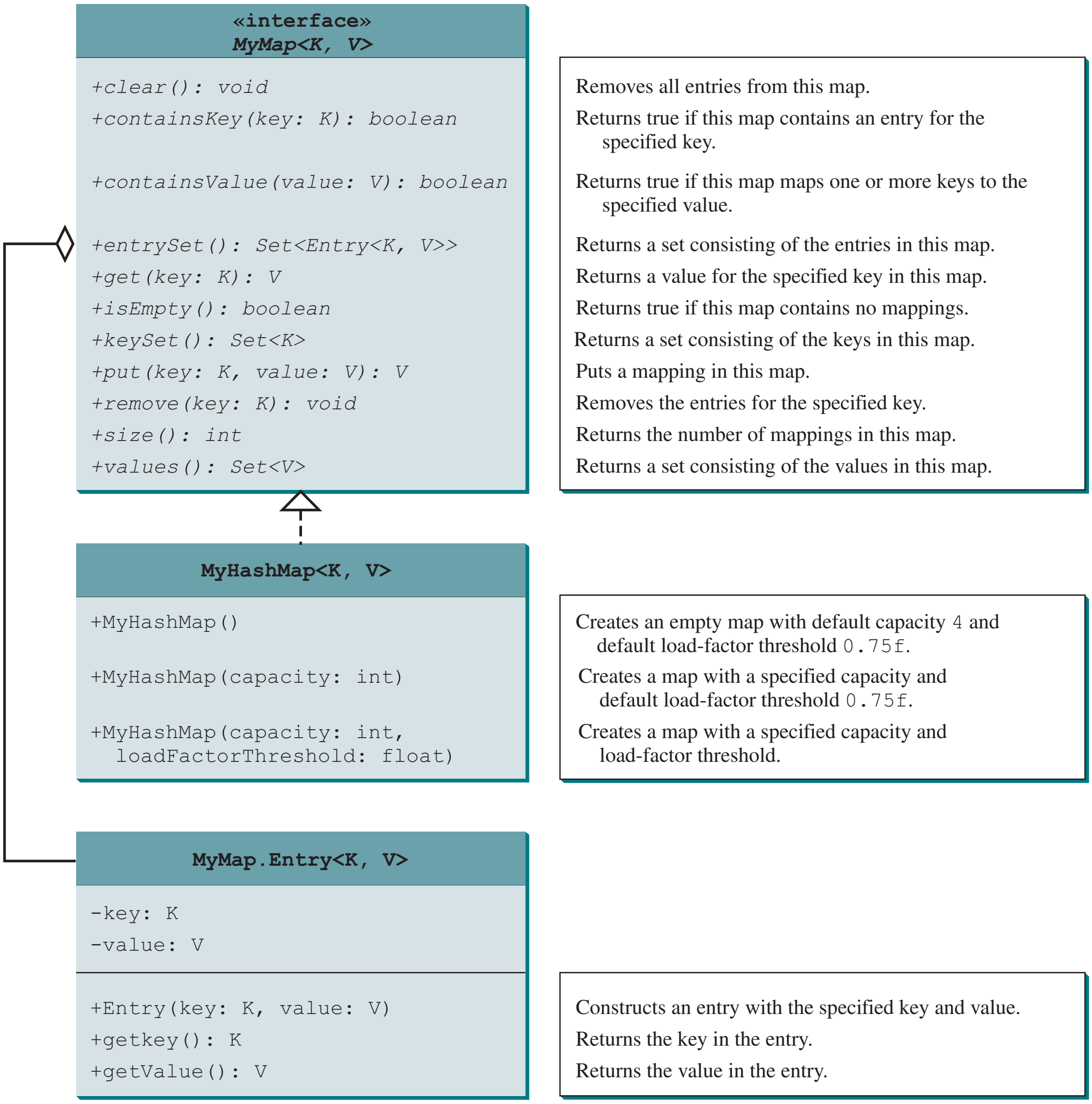
27.7 Implementing a Map Using Hashing

Key Point

A map can be implemented using hashing.

Now you understand the concept of hashing. You know how to design a good hash function to map a key to an index in a hash table, how to measure performance using the load factor, and how to increase the table size and rehash to maintain the performance. This section demonstrates how to implement a map using separate chaining.

We design our custom `Map` interface to mirror `java.util.Map` and name the interface `MyMap` and a concrete class `MyHashMap`, as shown in Figure 27.10.


«interface»
MyMap<K, V>
+clear(): void
+containsKey(key: K): boolean
+containsValue(value: V): boolean
+entrySet(): Set<Entry<K, V>>
+get(key: K): V
+isEmpty(): boolean
+keySet(): Set<K>
+put(key: K, value: V): V
+remove(key: K): void
+size(): int
+values(): Set<V>
MyHashMap<K, V>
+MyHashMap()
+MyHashMap(capacity: int)
+MyHashMap(capacity: int, loadFactorThreshold: float)
MyMap.Entry<K, V>
-key: K
-value: V
+Entry(key: K, value: V)
+getkey(): K
+getValue(): V

Removes all entries from this map.
Returns true if this map contains an entry for the specified key.
Returns true if this map maps one or more keys to the specified value.
Returns a set consisting of the entries in this map.
Returns a value for the specified key in this map.
Returns true if this map contains no mappings.
Returns a set consisting of the keys in this map.
Puts a mapping in this map.
Removes the entries for the specified key.
Returns the number of mappings in this map.
Returns a set consisting of the values in this map.

Creates an empty map with default capacity 4 and default load-factor threshold `0.75f`.
Creates a map with a specified capacity and default load-factor threshold `0.75f`.
Creates a map with a specified capacity and load-factor threshold.

Constructs an entry with the specified key and value.
Returns the key in the entry.
Returns the value in the entry.

FIGURE 27.10 **`MyHashMap`** implements the **`MyMap`** interface.

How do you implement **`MyHashMap`**? We will use an array for the hash table and each element in the hash table is a bucket. The bucket is a **`LinkedList`**. Listing 27.1 shows the **`MyMap`** interface and Listing 27.2 implements **`MyHashMap`** using separate chaining.

LISTING 27.1 MyMap.java

`interface MyMap`
`clear`
`containsKey`
`containsValue`

```java
public interface MyMap<K, V> {
  /** Remove all of the entries from this map */
  public void clear();

  /** Return true if the specified key is in the map */
  public boolean containsKey(K key);

  /** Return true if this map contains the specified value */
  public boolean containsValue(V value);

  /** Return a set of entries in the map */
  public java.util.Set<Entry<K, V>> entrySet();

  /** Return the value that matches the specified key */
  public V get(K key);

  /** Return true if this map doesn't contain any entries */
  public boolean isEmpty();

  /** Return a set consisting of the keys in this map */
  public java.util.Set<K> keySet();

  /** Add an entry (key, value) into the map */
  public V put(K key, V value);

  /** Remove an entry for the specified key */
  public void remove(K key);

  /** Return the number of mappings in this map */
  public int size();

  /** Return a set consisting of the values in this map */
  public java.util.Set<V> values();

  /** Define an inner class for Entry */
  public static class Entry<K, V> {
    K key;
    V value;

    public Entry(K key, V value) {
      this.key = key;
      this.value = value;
    }

    public K getKey() {
      return key;
    }

    public V getValue() {
      return value;
    }

    @Override
    public String toString() {
      return "[" + key + ", " + value + "]";
    }
  }
}
```

`entrySet`

`get`

`isEmpty`

`keySet`

`put`

`remove`

`size`

`values`

`Entry` inner class

LISTING 27.2 MyHashMap.java

```
import java.util.LinkedList;

public class MyHashMap<K, V> implements MyMap<K, V> {
  // Define the default hash-table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 4;

  // Define the maximum hash-table size. 1 << 30 is same as 2^30
  private static int MAXIMUM_CAPACITY = 1 << 30;
```

class `MyHashMap`

default initial capacity

maximum capacity

current capacity

default load factor

load-factor threshold

size

hash table

no-arg constructor

constructor

constructor

`clear`

`containsKey`

`containsValue`

```

  // Current hash-table capacity. Capacity is a power of 2
  private int capacity;

  // Define default load factor
  private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

  // Specify a load factor used in the hash table
  private float loadFactorThreshold;

  // The number of entries in the map
  private int size = 0;

  // Hash table is an array with each cell being a linked list
  LinkedList<MyMap.Entry<K,V>>[] table;

  /** Construct a map with the default capacity and load factor */
  public MyHashMap() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a map with the specified initial capacity and
   * default load factor */
  public MyHashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a map with the specified initial capacity
   * and load factor */
  public MyHashMap(int initialCapacity, float loadFactorThreshold) {
    if (initialCapacity > MAXIMUM_CAPACITY)
      this.capacity = MAXIMUM_CAPACITY;
    else
      this.capacity = trimToPowerOf2(initialCapacity);

    this.loadFactorThreshold = loadFactorThreshold;
    table = new LinkedList[capacity];
  }

  @Override /** Remove all of the entries from this map */
  public void clear() {
    size = 0;
    removeEntries();
  }

  @Override /** Return true if the specified key is in the map */
  public boolean containsKey(K key) {
    if (get(key) != null)
      return true;
    else
      return false;
  }

  @Override /** Return true if this map contains the value */
  public boolean containsValue(V value) {
    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        LinkedList<Entry<K, V>> bucket = table[i];
        for (Entry<K, V> entry: bucket)
          if (entry.getValue().equals(value))
```

```
            return true;
      }
    }

    return false;
  }

```

entrySet

```
  @Override /** Return a set of entries in the map */
  public java.util.Set<MyMap.Entry<K,V>> entrySet() {
    java.util.Set<MyMap.Entry<K, V>> set =
      new java.util.HashSet<>();

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        LinkedList<Entry<K, V>> bucket = table[i];
        for (Entry<K, V> entry: bucket)
          set.add(entry);
      }
    }

    return set;
  }

```

get

```
  @Override /** Return the value that matches the specified key */
  public V get(K key) {
    int bucketIndex = hash(key.hashCode());
    if (table[bucketIndex] != null) {
      LinkedList<Entry<K, V>> bucket = table[bucketIndex];
      for (Entry<K, V> entry: bucket)
        if (entry.getKey().equals(key))
          return entry.getValue();
    }

    return null;
  }

```

isEmpty

```
  @Override /** Return true if this map contains no entries */
  public boolean isEmpty() {
    return size == 0;
  }

```

keySet

```
  @Override /** Return a set consisting of the keys in this map */
  public java.util.Set<K> keySet() {
    java.util.Set<K> set = new java.util.HashSet<>();

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        LinkedList<Entry<K, V>> bucket = table[i];
        for (Entry<K, V> entry: bucket)
          set.add(entry.getKey());
      }
    }

    return set;
  }

```

put

```
  @Override /** Add an entry (key, value) into the map */
  public V put(K key, V value) {
    if (get(key) != null) { // The key is already in the map
      int bucketIndex = hash(key.hashCode());
      LinkedList<Entry<K, V>> bucket = table[bucketIndex];
      for (Entry<K, V> entry: bucket)
        if (entry.getKey().equals(key)) {
          V oldValue = entry.getValue();
          // Replace old value with new value
          entry.value = value;
          // Return the old value for the key
          return oldValue;
        }
    }

    // Check load factor
    if (size >= capacity * loadFactorThreshold) {
      if (capacity == MAXIMUM_CAPACITY)
        throw new RuntimeException("Exceeding maximum capacity");

      rehash();
    }

    int bucketIndex = hash(key.hashCode());

    // Create a linked list for the bucket if not already created
    if (table[bucketIndex] == null) {
      table[bucketIndex] = new LinkedList<Entry<K, V>>();
    }

    // Add a new entry (key, value) to hashTable[index]
    table[bucketIndex].add(new MyMap.Entry<K, V>(key, value));

    size++; // Increase size

    return value;
  }

  @Override /** Remove the entries for the specified key */
```

remove

```
  public void remove(K key) {
    int bucketIndex = hash(key.hashCode());

    // Remove the first entry that matches the key from a bucket
    if (table[bucketIndex] != null) {
      LinkedList<Entry<K, V>> bucket = table[bucketIndex];
      for (Entry<K, V> entry: bucket)
        if (entry.getKey().equals(key)) {
          bucket.remove(entry);
          size--; // Decrease size
          break; // Remove just one entry that matches the key
        }
    }
  }

  @Override /** Return the number of entries in this map */
```

size

```
  public int size() {
    return size;
  }

  @Override /** Return a set consisting of the values in this map */
```

values

```
  public java.util.Set<V> values() {
    java.util.Set<V> set = new java.util.HashSet<>();

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        LinkedList<Entry<K, V>> bucket = table[i];
        for (Entry<K, V> entry: bucket)
          set.add(entry.getValue());
      }
    }

    return set;
  }

```

hash

```
  /** Hash function */
  private int hash(int hashCode) {
    return supplementalHash(hashCode) & (capacity - 1);
  }

```

supplementalHash

```
  /** Ensure the hashing is evenly distributed */
  private static int supplementalHash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
  }

```

trimToPowerOf2

```
  /** Return a power of 2 for initialCapacity */
  private int trimToPowerOf2(int initialCapacity) {
    int capacity = 1;
    while (capacity < initialCapacity) {
      capacity <<= 1; // Same as capacity *= 2. <= is more efficient
    }

    return capacity;
  }

```

removeEntries

```
  /** Remove all entries from each bucket */
  private void removeEntries() {
    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        table[i].clear();
      }
    }
  }

```

rehash

```
  /** Rehash the map */
  private void rehash() {
    java.util.Set<Entry<K, V>> set = entrySet(); // Get entries
    capacity <<= 1; // Same as capacity *= 2. <= is more efficient
    table = new LinkedList[capacity]; // Create a new hash table
    size = 0; // Reset size to 0

    for (Entry<K, V> entry: set) {
      put(entry.getKey(), entry.getValue()); // Store to new table
    }
  }

```

toString

```
  @Override /** Return a string representation for this map */
  public String toString() {
    StringBuilder builder = new StringBuilder("[");

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null && table[i].size() > 0)
        for (Entry<K, V> entry: table[i])
          builder.append(entry);
```

```
249      }
250
251      builder.append("]");
252      return builder.toString();
253    }
254  }
```

hash-table parameters

The `MyHashMap` class implements the `MyMap` interface using separate chaining. The parameters that determine the hash-table size and load factors are defined in the class. The default initial capacity is `4` (line 5) and the maximum capacity is 2^{30} (line 8). The current hash-table capacity is designed as a value of the power of `2` (line 11). The default load-factor threshold is `0.75f` (line 14). You can specify a custom load-factor threshold when constructing a map. The custom load-factor threshold is stored in `loadFactorThreshold` (line 17). The data field `size` denotes the number of entries in the map (line 20). The hash table is an array. Each cell in the array is a linked list (line 23).

Three constructors

Three constructors are provided to construct a map. You can construct a default map with the default capacity and load-factor threshold using the no-arg constructor (lines 26–28), a map with the specified capacity and a default load-factor threshold (lines 32–34), and a map with the specified capacity and load-factor threshold (lines 38–46).

`clear`

The `clear` method removes all entries from the map (lines 49–52). It invokes `removeEntries()`, which deletes all entries in the buckets (lines 221–227). The `removeEntries()` method takes $O(capacity)$ time to clear all entries in the table.

`containsKey`

The `containsKey(key)` method checks whether the specified key is in the map by invoking the `get` method (lines 55–60). Since the `get` method takes $O(1)$ time, the `containsKey(key)` method takes $O(1)$ time.

`containsValue`

The `containsValue(value)` method checks whether the value is in the map (lines 63–74). This method takes $O(capacity + size)$ time. It is actually $O(capacity)$, since $capacity > size$.

`entrySet`

The `entrySet()` method returns a set that contains all entries in the map (lines 77–90). This method takes $O(capacity)$ time.

`get`

The `get(key)` method returns the value of the first entry with the specified key (lines 93–103). This method takes $O(1)$ time.

`isEmpty`

The `isEmpty()` method simply returns true if the map is empty (lines 106–108). This method takes $O(1)$ time.

`keySet`

The `keySet()` method returns all keys in the map as a set. The method finds the keys from each bucket and adds them to a set (lines 111–123). This method takes $O(capacity)$ time.

`put`

The `put(key, value)` method adds a new entry into the map. The method first tests if the key is already in the map (line 127), if so, it locates the entry and replaces the old value with the new value in the entry for the key (line 134) and the old value is returned (line 136). If the key is new in the map, the new entry is created in the map (line 156). Before inserting the new entry, the method checks whether the size exceeds the load-factor threshold (line 141). If so, the program invokes `rehash()` (line 145) to increase the capacity and store entries into a new larger hash table.

`rehash`

The `rehash()` method first copies all entries in a set (line 231), doubles the capacity (line 232), creates a new hash table (line 233), and resets the size to `0` (line 234). The method then copies the entries into the new hash table (lines 236–238). The `rehash` method takes $O(capacity)$ time. If no rehash is performed, the `put` method takes $O(1)$ time to add a new entry.

`remove`

The `remove(key)` method removes the entry with the specified key in the map (lines 164–177). This method takes $O(1)$ time.

`size`

The `size()` method simply returns the size of the map (lines 180–182). This method takes $O(1)$ time.

`values`

The `values()` method returns all values in the map. The method examines each entry from all buckets and adds it to a set (lines 185–197). This method takes $O(capacity)$ time.

`hash`

The `hash()` method invokes the `supplementalHash` to ensure the hashing is evenly distributed to produce an index for the hash table (lines 200–208). This method takes $O(1)$ time.

Table 27.1 summarizes the time complexities of the methods in **MyHashMap**.

Since rehashing does not happen very often, the time complexity for the **put** method is *O*(1). Note the complexities of the **clear**, **entrySet**, **keySet**, **values**, and **rehash** methods depend on **capacity**, so to avoid poor performance for these methods, you should choose an initial capacity carefully.

TABLE 27.1 Time Complexities for Methods in **MyHashMap**

Methods	*Time*
`clear()`	*O*(*capacity*)
`containsKey(key: Key)`	*O*(1)
`containsValue(value: V)`	*O*(*capacity*)
`entrySet()`	*O*(*capacity*)
`get(key: K)`	*O*(1)
`isEmpty()`	*O*(1)
`keySet()`	*O*(*capacity*)
`put(key: K, value: V)`	*O*(1)
`remove(key: K)`	*O*(1)
`size()`	*O*(1)
`values()`	*O*(*capacity*)
`rehash()`	*O*(*capacity*)

Listing 27.3 gives a test program that uses **MyHashMap**.

LISTING 27.3 TestMyHashMap.java

```java
public class TestMyHashMap {
  public static void main(String[] args) {
    // Create a map
    MyMap<String, Integer> map = new MyHashMap<>();
    map.put("Smith", 30);
    map.put("Anderson", 31);
    map.put("Lewis", 29);
    map.put("Cook", 29);
    map.put("Smith", 65);

    System.out.println("Entries in map: " + map);

    System.out.println("The age for Lewis is " +
      map.get("Lewis"));

    System.out.println("Is Smith in the map? " +
      map.containsKey("Smith"));
    System.out.println("Is age 33 in the map? " +
      map.containsValue(33));

    map.remove("Smith");
    System.out.println("Entries in map: " + map);

    map.clear();
    System.out.println("Entries in map: " + map);
  }
}
```

create a map

put entries

display entries

get value

is key in map?

is value in map?

remove entry

```
Entries in map: [[Anderson, 31][Smith, 65][Lewis, 29][Cook, 29]]
The age for Lewis is 29
Is Smith in the map? true
Is age 33 in the map? false
Entries in map: [[Anderson, 31][Lewis, 29][Cook, 29]]
Entries in map: []
```

The program creates a map using `MyHashMap` (line 4) and adds five entries into the map (lines 5–9). Line 5 adds key `Smith` with value `30` and line 9 adds `Smith` with value `65`. The latter value replaces the former value. The map actually has only four entries. The program displays the entries in the map (line 11), gets a value for a key (line 14), checks whether the map contains the key (line 17) and a value (line 19), removes an entry with the key `Smith` (line 21), and redisplays the entries in the map (line 22). Finally, the program clears the map (line 24) and displays an empty map (line 25).

Check Point

27.7.1 What is `1 << 30` in line 8 in Listing 27.2? What are the integers resulted from `1 << 1`, `1 << 2`, and `1 << 3`?

27.7.2 What are the integers resulted from `32 >> 1`, `32 >> 2`, `32 >> 3`, and `32 >> 4`?

27.7.3 In Listing 27.2, will the program work if `LinkedList` is replaced by `ArrayList`? In Listing 27.2, how do you replace the code in lines 56–59 using one line of code?

27.7.4 Describe how the `put(key, value)` method is implemented in the `MyHashMap` class.

27.7.5 In Listing 27.2, the `supplementalHash` method is declared static. Can the `hash` method be declared static?

27.7.6 Show the output of the following code:

```
MyMap<String, String> map = new MyHashMap<>();
map.put("Texas", "Dallas");
map.put("Oklahoma", "Norman");
map.put("Texas", "Austin");
map.put("Oklahoma", "Tulsa");

System.out.println(map.get("Texas"));
System.out.println(map.size());
```

27.7.7 If `x` is a negative `int` value, will `x & (N – 1)` be negative?

hash set
hash map
set

27.8 Implementing Set Using Hashing

A hash set can be implemented using a hash map.

Key Point

A *set* (introduced in Chapter 21) is a data structure that stores distinct values. The Java Collections Framework defines the `java.util.Set` interface for modeling sets. Three concrete implementations are `java.util.HashSet`, `java.util.LinkedHashSet`, and `java.util.TreeSet`. `java.util.HashSet` is implemented using hashing, `java.util.LinkedHashSet` using `LinkedList`, and `java.util.TreeSet` using binary search trees.

You can implement `MyHashSet` using the same approach as for implementing `MyHashMap`. The only difference is that key/value pairs are stored in the map, while elements are stored in the set.

`MyHashSet`

Since all the methods in `HashSet` are inherited from `Collection`, we design our custom `HashSet` by implementing the `Collection` interface, as shown in Figure 27.11.

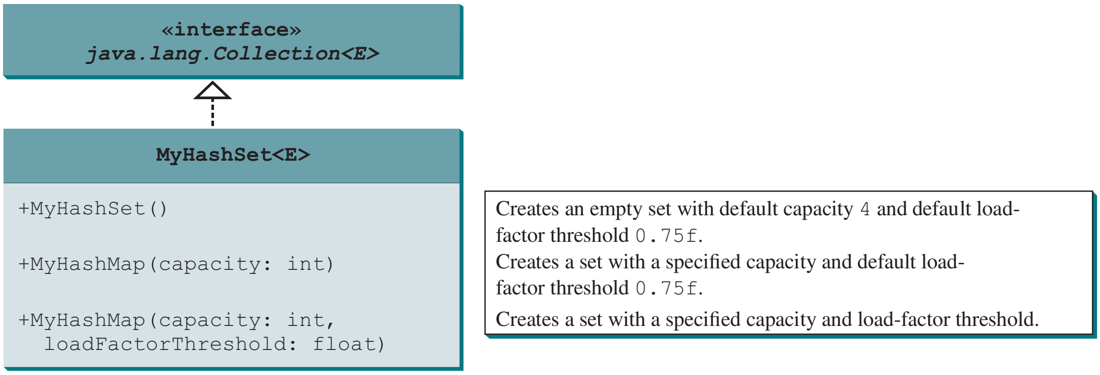

«interface»
java.lang.Collection<E>
MyHashSet<E>
+MyHashSet()
+MyHashMap(capacity: int)
+MyHashMap(capacity: int, loadFactorThreshold: float)
Creates an empty set with default capacity 4 and default load-factor threshold 0.75f.
Creates a set with a specified capacity and default load-factor threshold 0.75f.
Creates a set with a specified capacity and load-factor threshold.

FIGURE 27.11 `MyHashSet` implements the `Collection` interface.

Listing 27.4 implements `MyHashSet` using separate chaining.

LISTING 27.4 MyHashSet.java

```java
import java.util.*;

public class MyHashSet<E> implements Collection<E> {
  // Define the default hash-table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 4;

  // Define the maximum hash-table size. 1 << 30 is same as 2^30
  private static int MAXIMUM_CAPACITY = 1 << 30;

  // Current hash-table capacity. Capacity is a power of 2
  private int capacity;

  // Define default load factor
  private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

  // Specify a load-factor threshold used in the hash table
  private float loadFactorThreshold;

  // The number of elements in the set
  private int size = 0;

  // Hash table is an array with each cell being a linked list
  private LinkedList<E>[] table;

  /** Construct a set with the default capacity and load factor */
  public MyHashSet() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a set with the specified initial capacity and
   * default load factor */
  public MyHashSet(int initialCapacity) {
    this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a set with the specified initial capacity
   * and load factor */
  public MyHashSet(int initialCapacity, float loadFactorThreshold) {
```

class `MyHashSet`
default initial capacity
maximum capacity
current capacity
default max load factor
load-factor threshold
size
hash table
no-arg constructor
constructor
constructor

```
    if (initialCapacity > MAXIMUM_CAPACITY)
      this.capacity = MAXIMUM_CAPACITY;
    else
      this.capacity = trimToPowerOf2(initialCapacity);

    this.loadFactorThreshold = loadFactorThreshold;
    table = new LinkedList[capacity];
  }

  @Override /** Remove all elements from this set */
```

clear

```
  public void clear() {
    size = 0;
    removeElements();
  }

  @Override /** Return true if the element is in the set */
```

contains

```
  public boolean contains(E e) {
    int bucketIndex = hash(e.hashCode());
    if (table[bucketIndex] != null) {
      LinkedList<E> bucket = table[bucketIndex];
      return bucket.contains(e);
    }

    return false;
  }

  @Override /** Add an element to the set */
```

add

```
  public boolean add(E e) {
    if (contains(e))  // Duplicate element not stored
      return false;

    if (size + 1 > capacity * loadFactorThreshold) {
      if (capacity == MAXIMUM_CAPACITY)
        throw new RuntimeException("Exceeding maximum capacity");

      rehash();
    }

    int bucketIndex = hash(e.hashCode());

    // Create a linked list for the bucket if not already created
    if (table[bucketIndex] == null) {
      table[bucketIndex] = new LinkedList<E>();
    }

    // Add e to hashTable[index]
    table[bucketIndex].add(e);

    size++; // Increase size

    return true;
  }

  @Override /** Remove the element from the set */
```

remove

```
  public boolean remove(E e) {
    if (!contains(e))
      return false;

    int bucketIndex = hash(e.hashCode());

```

```
      // Create a linked list for the bucket if not already created
      if (table[bucketIndex] != null) {
        LinkedList<E> bucket = table[bucketIndex];
        bucket.removed(e);
      }

      size--; // Decrease size

      return true;
    }

    @Override /** Return true if the set contain no elements */
    public boolean isEmpty() {
      return size == 0;
    }

    @Override /** Return the number of elements in the set */
    public int size() {
      return size;
    }

    @Override /** Return an iterator for the elements in this set */
    public java.util.Iterator<E> iterator() {
      return new MyHashSetIterator(this);
    }

    /** Inner class for iterator */
    private class MyHashSetIterator implements java.util.Iterator<E> {
      // Store the elements in a list
      private java.util.ArrayList<E> list;
      private int current = 0; // Point to the current element in list
      private MyHashSet<E> set;

      /** Create a list from the set */
      public MyHashSetIterator(MyHashSet<E> set) {
        this.set = set;
        list = setToList();
      }

      @Override /** Next element for traversing? */
      public boolean hasNext() {
        return current < list.size();
      }

      @Override /** Get current element and move cursor to the next */
      public E next() {
        return list.get(current++);
      }

      /** Remove the current element returned by the last next() */
      public void remove() {
        // Left as an exercise
        // You need to remove the element from the set
        // You also need to remove it from the list
      }
    }

    /** Hash function */
    private int hash(int hashCode) {
      return supplementalHash(hashCode) & (capacity - 1);
```

`isEmpty`

`size`

`iterator`

inner class

`hash`

```
  }

  /** Ensure the hashing is evenly distributed */
```

supplementalHash

```
  private static int supplementalHash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
  }

  /** Return a power of 2 for initialCapacity */
```

trimToPowerOf2

```
  private int trimToPowerOf2(int initialCapacity) {
    int capacity = 1;
    while (capacity < initialCapacity) {
      capacity <<= 1; // Same as capacity *= 2. <= is more efficient
    }

    return capacity;
  }

  /** Remove all e from each bucket */
  private void removeElements() {
    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        table[i].clear();
      }
    }
  }

  /** Rehash the set */
```

rehash

```
  private void rehash() {
    java.util.ArrayList<E> list = setToList(); // Copy to a list
    capacity <<= 1; // Same as capacity *= 2. <= is more efficient
    table = new LinkedList[capacity]; // Create a new hash table
    size = 0;

    for (E element: list) {
      add(element); // Add from the old table to the new table
    }
  }

  /** Copy elements in the hash set to an array list */
```

setToList

```
  private java.util.ArrayList<E> setToList() {
    java.util.ArrayList<E> list = new java.util.ArrayList<>();

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        for (E e: table[i]) {
          list.add(e);
        }
      }
    }

    return list;
  }

  @Override /** Return a string representation for this set */
```

toString

```
  public String toString() {
    java.util.ArrayList<E> list = setToList();
    StringBuilder builder = new StringBuilder("[");

    // Add the elements except the last one to the string builder
    for (int i = 0; i < list.size() - 1; i++) {
      builder.append(list.get(i) + ", ");
    }

    // Add the last element in the list to the string builder
    if (list.size() == 0)
      builder.append("]");
    else
      builder.append(list.get(list.size() - 1) + "]");

    return builder.toString();
  }

  @Override
  public boolean addAll(Collection<? extends E> arg0) {
    // Left as an exercise
    return false;
  }

  @Override
  public boolean containsAll(Collection<?> arg0) {
    // Left as an exercise
    return false;
  }

  @Override
  public boolean removeAll(Collection<?> arg0) {
    // Left as an exercise
    return false;
  }

  @Override
  public boolean retainAll(Collection<?> arg0) {
    // Left as an exercise
    return false;
  }

  @Override
  public Object[] toArray() {
    // Left as an exercise
    return null;
  }

  @Override
  public <T> T[] toArray(T[] arg0) {
    // Left as an exercise
    return null;
  }
}
```

override `addAll`

override `containsAll`

override `removeAll`

override `retainAll`

override `toArray()`

`MyHashSet` vs. `MyHashMap`

The **`MyHashSet`** class implements the **`MySet`** interface using separate chaining. Implementing **`MyHashSet`** is very similar to implementing **`MyHashMap`** except for the following differences:

1. The elements are stored in the hash table for **`MyHashSet`**, but the entries (key/value pairs) are stored in the hash table for **`MyHashMap`**.
2. **`MyHashSet`** implements **`Collection`**. Since **`Collection`** implements **`Iterable`**, the elements in **`MyHashSet`** are iterable.

three constructors

Three constructors are provided to construct a set. You can construct a default set with the default capacity and load factor using the no-arg constructor (lines 26–28), a set with the specified capacity and a default load factor (lines 32–34), and a set with the specified capacity and load factor (lines 38–46).

`clear`

The `clear` method removes all elements from the set (lines 49–52). It invokes `removeElements()`, which clears all table cells (line 181). Each table cell is a linked list that stores the elements with the same hash table index. The `removeElements()` method takes *O*(*capacity*) time.

`contains`

The `contains(element)` method checks whether the specified element is in the set by examining whether the designated bucket contains the element (line 59). This method takes *O*(1) time because the bucket size is considered very small.

`add`

The `add(element)` method adds a new element into the set. The method first checks if the element is already in the set (line 67). If so, the method returns false. The method then checks whether the size exceeds the load-factor threshold (line 70). If so, the program invokes `rehash()` (line 74) to increase the capacity and store elements into a new larger hash table.

`rehash`

The `rehash()` method first copies all elements to a list (line 188), doubles the capacity (line 189), creates a new hash table (line 190), and resets the size to `0` (line 191). The method then copies the elements into the new larger hash table (lines 193–195). The `rehash` method takes *O*(*capacity*) time. If no rehash is performed, the `add` method takes *O*(1) time to add a new element.

`remove`

The `remove(element)` method removes the specified element in the set (lines 93–108). This method takes *O*(1) time.

`size`

The `size()` method simply returns the number of elements in the set (lines 116–118). This method takes *O*(1) time.

`iterator`

The `iterator()` method returns an instance of `java.util.Iterator`. The `MyHashSetIterator` class implements `java.util.Iterator` to create a forward iterator. When a `MyHashSetIterator` is constructed, it copies all the elements in the set to a list (line 135). The variable `current` points to the element in the list. Initially, `current` is `0` (line 129), which points to the first element in the list. `MyHashSetIterator` implements the methods `hasNext()`, `next()`, and `remove()` in `java.util.Iterator`. Invoking `hasNext()` returns true if `current < list.size()`. Invoking `next()` returns the current element and moves `current` to point to the next element (line 145). Invoking `remove()` removes the element called by the last `next()`.

`hash`

The `hash()` method invokes the `supplementalHash` to ensure the hashing is evenly distributed to produce an index for the hash table (lines 157–159). This method takes *O*(1) time.

The methods `containsAll`, `addAll`, `removeAll`, `retainAll`, `toArray()`, and `toArray(T[])` defined in the `Collection` interface are overridden in `MyHashSet`. Their implementations are left as exercises in Programming Exercise 27.11.

Table 27.2 summarizes the time complexity of the methods in `MyHashSet`.

TABLE 27.2 Time Complexities for Methods in `MyHashSet`

Methods	*Time*
`clear()`	*O*(*capacity*)
`contains(e: E)`	*O*(1)
`add(e: E)`	*O*(1)
`remove(e: E)`	*O*(1)
`isEmpty()`	*O*(1)
`size()`	*O*(1)
`iterator()`	*O*(*capacity*)
`rehash()`	*O*(*capacity*)

Listing 27.5 gives a test program that uses `MyHashSet`.

LISTING 27.5 TestMyHashSet.java

```
 1  public class TestMyHashSet {
 2    public static void main(String[] args) {
 3      // Create a MyHashSet
 4      java.util.Collection<String> set = new MyHashSet<>();
 5      set.add("Smith");
 6      set.add("Anderson");
 7      set.add("Lewis");
 8      set.add("Cook");
 9      set.add("Smith");
10
11      System.out.println("Elements in set: " + set);
12      System.out.println("Number of elements in set: " + set.size());
13      System.out.println("Is Smith in set? " + set.contains("Smith"));
14
15      set.remove("Smith");
16      System.out.print("Names in set in uppercase are ");
17      for (String s: set)
18        System.out.print(s.toUpperCase() + " ");
19
20      set.clear();
21      System.out.println("\nElements in set: " + set);
22    }
23  }
```

create a set
add elements
display elements
set size
remove element
foreach loop
clear set

```
Elements in set: [Cook, Anderson, Smith, Lewis]
Number of elements in set: 4
Is Smith in set? true
Names in set in uppercase are COOK ANDERSON LEWIS
Elements in set: []
```

The program creates a set using `MyHashSet` (line 4) and adds five elements to the set (lines 5–9). Line 5 adds `Smith` and line 9 adds `Smith` again. Since only nonduplicate elements are stored in the set, `Smith` appears in the set only once. The set actually has four elements. The program displays the elements (line 11), gets its size (line 12), checks whether the set contains a specified element (line 13), and removes an element (line 15). Since the elements in a set are iterable, a foreach loop is used to traverse all elements in the set (lines 17–18). Finally, the program clears the set (line 20) and displays an empty set (line 21).

Check Point

27.8.1 Why can you use a foreach loop to traverse the elements in a set?

27.8.2 Describe how the `add(e)` method is implemented in the `MyHashSet` class.

27.8.3 Can lines 100–103 in Listing 27.4 be removed?

27.8.4 Implement the `remove()` method in lines 150–152?

KEY TERMS

associative array	1016	hash function	1016
cluster	1020	hash map	1034
dictionary	1016	hash set	1034
double hashing	1022	hash table	1016
hash code	1017	linear probing	1019

load factor	1025
open addressing	1019
perfect hash function	1016
polynomial hash code	1018
quadratic probing	1020
rehashing	1025
separate chaining	1023

CHAPTER SUMMARY

1. A *map* is a data structure that stores entries. Each entry contains two parts: a *key* and a *value*. The key is also called a *search key*, which is used to search for the corresponding value. You can implement a map to obtain $O(1)$ time complexity on searching, retrieval, insertion, and deletion using the hashing technique.

2. A *set* is a data structure that stores elements. You can use the hashing technique to implement a set to achieve $O(1)$ time complexity on searching, insertion, and deletion for a set.

3. *Hashing* is a technique that retrieves the value using the index obtained from a key without performing a search. A typical *hash function* first converts a search key to an integer value called a *hash code*, then compresses the hash code into an index to the *hash table*.

4. A *collision* occurs when two keys are mapped to the same index in a hash table. Generally, there are two ways for handling collisions: *open addressing* and *separate chaining*.

5. Open addressing is the process of finding an open location in the hash table in the event of collision. Open addressing has several variations: *linear probing*, *quadratic probing*, and *double hashing*.

6. The *separate chaining* scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a *bucket*. A bucket is a container that holds multiple entries.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

****27.1** (*Implement `MyMap` using open addressing with linear probing*) Create a new concrete class that implements `MyMap` using open addressing with linear probing. For simplicity, use `f(key) = key % size` as the hash function, where size is the hash-table size. Initially, the hash-table size is `4`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`). Test your new `MyHashMap` class using the code at https://liveexample.pearsoncmg.com/test/Exercise27_01.txt.

****27.2** (*Implement `MyMap` using open addressing with quadratic probing*) Create a new concrete class that implements `MyMap` using open addressing with quadratic probing. For simplicity, use `f(key) = key % size` as the hash function, where `size` is the hash-table size. Initially, the hash-table size is `4`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`).

****27.3** (*Implement `MyMap` using open addressing with double hashing*) Create a new concrete class that implements `MyMap` using open addressing with double hashing. For simplicity, use `f(key) = key % size` as the hash function, where `size` is the hash-table size. Initially, the hash-table size is `4`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`).

****27.4** (*Modify `MyHashMap` with duplicate keys*) Modify `MyHashMap` to allow duplicate keys for entries. You need to modify the implementation for the `put(key, value)` method. Also add a new method named `getAll(key)` that returns a set of values that match the key in the map.

****27.5** (*Implement `MyHashSet` using `MyHashMap`*) Implement `MyHashSet` using `MyHashMap`. Note you can create entries with (`key, key`), rather than (`key, value`).

****27.6** (*Animate linear probing*) Write a program that animates linear probing, as shown in Figure 27.3. You can change the initial size of the hash table in the program. Assume the load-factor threshold is `0.75`.

****27.7** (*Animate separate chaining*) Write a program that animates `MyHashMap`, as shown in Figure 27.9. You can change the initial size of the table. Assume the load-factor threshold is `0.75`.

****27.8** (*Animate quadratic probing*) Write a program that animates quadratic probing, as shown in Figure 27.5. You can change the initial size of the hash table in program. Assume the load-factor threshold is `0.75`.

****27.9** (*Implement hashCode for string*) Write a method that returns a hash code for string using the approach described in Section 27.3.2 with `b` value `31`. The function header is as follows:

```
public static int hashCodeForString(String s)
```

****27.10** (*Compare `MyHashSet` and `MyArrayList`*) `MyArrayList` is defined in Listing 24.2. Write a program that generates `1000000` random double values between `0` and `999999` and stores them in a `MyArrayList` and in a `MyHashSet`. Generate a list of `1000000` random double values between `0` and `1999999`. For each number in the list, test if it is in the array list and in the hash set. Run your program to display the total test time for the array list and for the hash set.

****27.11** (*Implement set operations in `MyHashSet`*) The implementations of the methods `addAll`, `removeAll`, `retainAll`, `toArray()`, and `toArray(T[])` are omitted in the `MyHashSet` class. Implement these methods. Also add a new constructor `MyHashSet(E[] list)` in the `MyHashSet` class. Test your new `MyHashSet` class using the code at https://liveexample.pearsoncmg.com/test/Exercise27_11.txt.

****27.12** (`setToList`) Write the following method that returns an `ArrayList` from a set:

```
public static <E> ArrayList<E> setToList(Set<E> s)
```

***27.13** (*The `Date` class*) Design a class named `Date` that meets the following requirements:

- Three data fields `year`, `month`, and `day` for representing a date
- A constructor that constructs a date with the specified year, month, and day
- Override the `equals` method
- Override the `hashCode` method. (For reference, see the implementation of the `Date` class in the Java API.)

***27.14** (*The `Point` class*) Design a class named `Point` that meets the following requirements:

- Two data fields `x` and `y` for representing a point with getter methods
- A no-arg constructor that constructs a point for (`0`, `0`)

- A constructor that constructs a point with the specified **`x`** and **`y`** values
- Override the **`equals`** method. Point **`p1`** is said to be equal to point **`p2`** if `p1.x == p2.x` and `p1.y == p2.y`.
- Override the **`hashCode`** method. (For reference, see the implementation of the **`Point2D`** class in the Java API.)

***27.15** (*Modify Listing 27.4 MyHashSet.java*) The book uses **`LinkedList`** for buckets. Replace **`LinkedList`** with **`AVLTree`**. Assume **`E`** is **`Comparable`**. Redefine **`MyHashSet`** as follows:

```
public class MyHashSet<E extends Comparable<E>> implements
    Collection<E> {
...
}
```

Test your program using the main method in Listing 27.5.

CHAPTER 28

GRAPHS AND APPLICATIONS

Objectives

- To model real-world problems using graphs and explain the Seven Bridges of Königsberg problem (§28.1).
- To describe the graph terminologies: vertices, edges, simple graphs, weighted/unweighted graphs, and directed/undirected graphs (§28.2).
- To represent vertices and edges using lists, edge arrays, edge objects, adjacency matrices, and adjacency lists (§28.3).
- To model graphs using the `Graph` interface and the `UnweightedGraph` class (§28.4).
- To display graphs visually (§28.5).
- To represent the traversal of a graph using the `UnweightedGraph.SearchTree` class (§28.6).
- To design and implement depth-first search (§28.7).
- To solve the connected-circle problem using depth-first search (§28.8).
- To design and implement breadth-first search (§28.9).
- To solve the nine tails problem using breadth-first search (§28.10).

28.1 Introduction

Key Point

Many real-world problems can be solved using graph algorithms.

problem

Graphs are useful in modeling and solving real-world problems. For example, the problem to find the least number of flights between two cities can be modeled using a graph, where the vertices represent cities and the edges represent the flights between two adjacent cities, as shown in Figure 28.1. The problem of finding the minimal number of connecting flights between two cities is reduced to finding the shortest path between two vertices in a graph. At United Parcel Service, each driver makes an average 120 stops per day. There are many possible ways for ordering these stops. UPS spent hundreds of millions of dollars for 10 years to develop a system called Orion, which stands for On-Road Integrated Optimization and Navigation. The system uses graph algorithms to plan the most cost-efficient routes for each driver. This chapter studies the algorithms for unweighted graphs, and the next chapter studies those for weighted graphs.

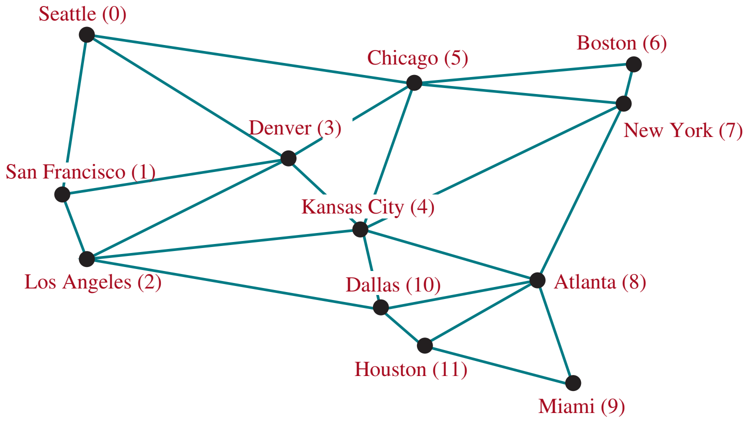
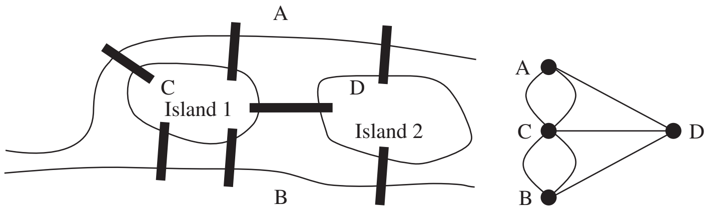


FIGURE 28.1 A graph can be used to model the flights between the cities.

graph theory

Seven Bridges of Königsberg

The study of graph problems is known as *graph theory*. Graph theory was founded by Leonhard Euler in 1736, when he introduced graph terminology to solve the famous *Seven Bridges of Königsberg* problem. The city of Königsberg, Prussia (now Kaliningrad, Russia), was divided by the Pregel River. There were two islands on the river. The city and islands were connected by seven bridges, as shown in Figure 28.2a. The question is, can one take a walk, cross each bridge exactly once, and return to the starting point? Euler proved that it is not possible.



(a) Seven bridges sketch (b) Graph model

FIGURE 28.2 Seven bridges connected islands and land.

To establish a proof, Euler first abstracted the Königsberg city map by eliminating all streets, producing the sketch shown in Figure 28.2a. Next, he replaced each land mass with a dot, called a *vertex* or a *node*, and each bridge with a line, called an *edge*, as shown in Figure 28.2b. This structure with vertices and edges is called a *graph*.

Looking at the graph, we ask whether there is a path starting from any vertex, traversing all edges exactly once, and returning to the starting vertex. Euler proved that for such a path to exist, each vertex must have an even number of edges. Therefore, the Seven Bridges of Königsberg problem has no solution.

Graphs have many applications in various areas, such as in computer science, mathematics, biology, engineering, economics, genetics, and social sciences. This chapter presents the algorithms for depth-first search and breadth-first search, and their applications. The next chapter presents the algorithms for finding a minimum spanning tree and shortest paths in weighted graphs, and their applications.

28.2 Basic Graph Terminologies

A graph consists of vertices and edges that connect the vertices.

Key Point

This chapter does not assume that you have any prior knowledge of graph theory or discrete mathematics. We use plain and simple terms to define graphs.

what is a graph?

What is a graph? A *graph* is a mathematical structure that represents relationships among entities in the real world. For example, the graph in Figure 28.1 represents the flights among cities, and the graph in Figure 28.2b represents the bridges among land masses.

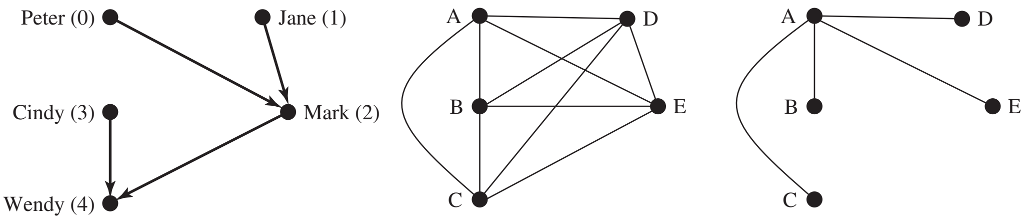
define a graph

A graph consists of a set of vertices (also known as *nodes* or *points*), and a set of edges that connect the vertices. For convenience, we define a graph as G = (V, E), where V represents a set of vertices and E represents a set of edges. For example, V and E for the graph in Figure 28.1 are as follows:

```
V = {"Seattle", "San Francisco", "Los Angeles",
     "Denver", "Kansas City", "Chicago", "Boston", "New York",
     "Atlanta", "Miami", "Dallas", "Houston"};

E = {{"Seattle", "San Francisco"},{"Seattle", "Chicago"},
     {"Seattle", "Denver"}, {"San Francisco", "Denver"},
      ...
    };
```

directed vs. undirected graphs

A graph may be directed or undirected. In a *directed graph*, each edge has a direction, which indicates you can move from one vertex to the other through the edge. You can model parent/child relationships using a directed graph, where an edge from vertex A to B indicates that A is a parent of B. Figure 28.3a shows a directed graph.



(a) A directed graph (b) A complete graph (c) A subgraph of the graph in (b)

Figure 28.3 Graphs may appear in many forms.

In an *undirected graph*, you can move in both directions between vertices. The graph in Figure 28.1 is undirected.

weighted vs. unweighted graphs

Edges may be weighted or unweighted. For example, you can assign a weight for each edge in the graph in Figure 28.1 to indicate the flight time between the two cities.

adjacent vertices
incident edges
degree

Two vertices in a graph are said to be *adjacent* if they are connected by the same edge. Similarly, two edges are said to be *adjacent* if they are connected to the same vertex. An edge in a graph that joins two vertices is said to be *incident* to both vertices. The *degree* of a vertex is the number of edges incident to it.

neighbor

Two vertices are *neighbors* if they are adjacent. Similarly, two edges are *neighbors* if they are adjacent.

loop
parallel edge
simple graph
complete graph

A *loop* is an edge that links a vertex to itself. If two vertices are connected by two or more edges, these edges are called *parallel edges*. A *simple graph* is one that doesn't have any loops or parallel edges. In a *complete graph*, every two vertices are adjacent, as shown in Figure 28.3b.

connected graph
subgraph

A graph is *connected* (also known as *strongly connected*) if there exists a path between every two vertices in the graph. A graph is *weakly connected* if it is connected when considering the graph is undirected. A *subgraph* of a graph *G* is a graph whose vertex set is a subset of that of *G* and whose edge set is a subset of that of *G*. For example, the graph in Figure 28.3c is a subgraph of the graph in Figure 28.3b.

cycle
tree
spanning tree

Assume the graph is connected and undirected. A *cycle* is a closed path that starts from a vertex and ends at the same vertex. A connected graph is a *tree* if it does not have cycles. A *spanning tree* of a graph *G* is a connected subgraph of *G*, and the subgraph is a tree that contains all vertices in *G*.

graph learning tool on Companion Website

Pedagogical Note

Before we introduce graph algorithms and applications, it is helpful to get acquainted with graphs using the interactive tool at liveexample.pearsoncmg.com/dsanimation /GraphLearning-TooleBook.html, as shown in Figure 28.4. The tool allows you to add/remove/move vertices and draw edges using mouse gestures. You can also find depth-first search (DFS) trees and breadth-first search (BFS) trees, and the shortest path between two vertices.

Check Point

28.2.1 What is the famous *Seven Bridges of Königsberg* problem?

28.2.2 What is a graph? Explain the following terms: undirected graph, directed graph, weighted graph, degree of a vertex, parallel edge, simple graph, complete graph, connected graph, cycle, subgraph, tree, and spanning tree.

28.2.3 How many edges are in a complete graph with 5 vertices? How many edges are in a tree of 5 vertices?

28.2.4 How many edges are in a complete graph with *n* vertices? How many edges are in a tree of *n* vertices?

28.3 Representing Graphs

Key Point

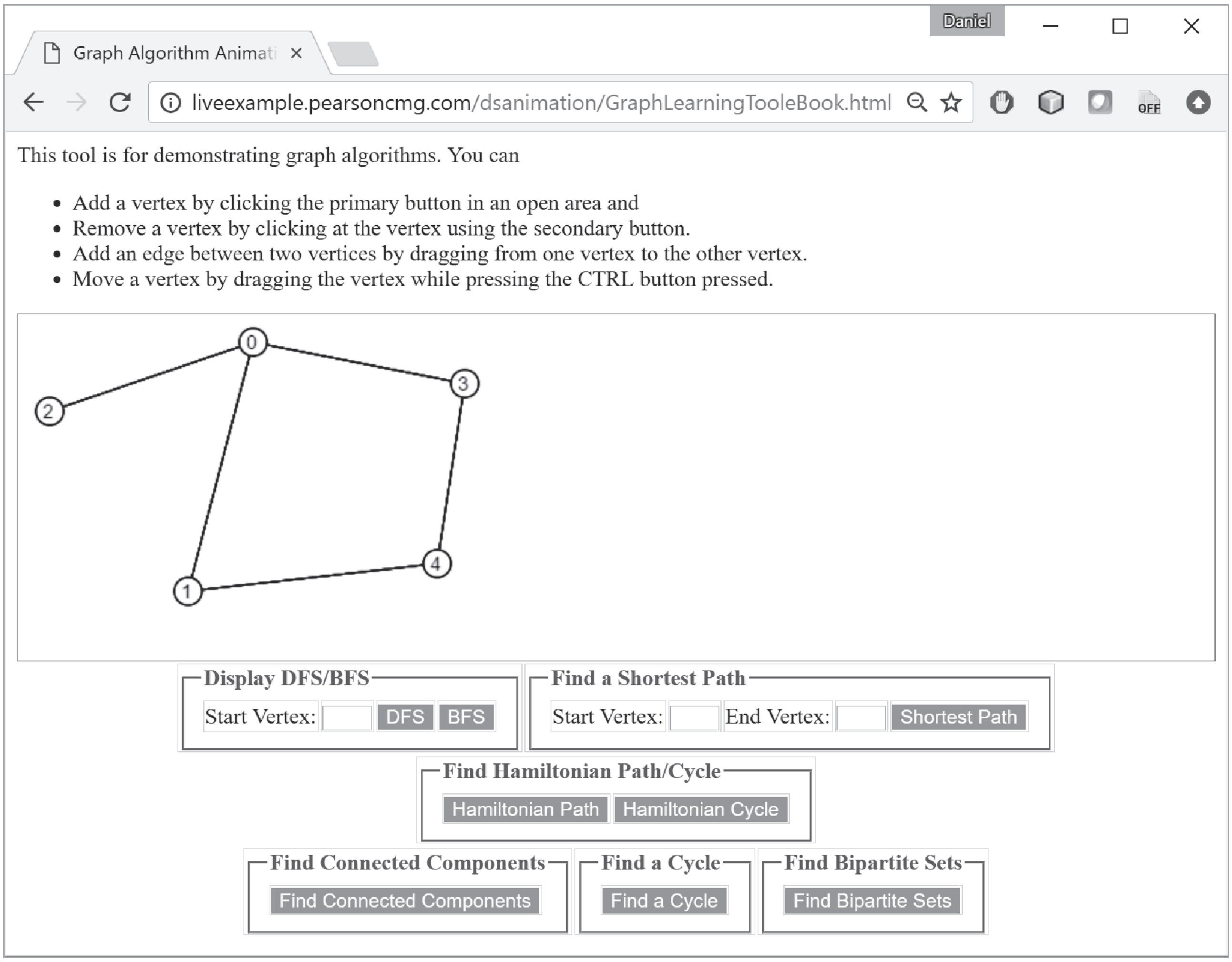
Representing a graph is to store its vertices and edges in a program. The data structure for storing a graph is arrays or lists.

To write a program that processes and manipulates graphs, you have to store or represent data for the graphs in the computer.

28.3.1 Representing Vertices

The vertices can be stored in an array or a list. For example, you can store all the city names in the graph in Figure 28.1 using the following array:

```
String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
  "Denver", "Kansas City", "Chicago", "Boston", "New York",
  "Atlanta", "Miami", "Dallas", "Houston"};
```


Daniel
Graph Algorithm Animati ×
liveexample.pearsoncmg.com/dsanimation/GraphLearningToolеBook.html
OFF
0
1
2
3
4
Display DFS/BFS
Start Vertex:
DFS
BFS
Find a Shortest Path
Start Vertex:
End Vertex:
Shortest Path
Find Hamiltonian Path/Cycle
Hamiltonian Path
Hamiltonian Cycle
Find Connected Components
Find Connected Components
Find a Cycle
Find a Cycle
Find Bipartite Sets
Find Bipartite Sets

This tool is for demonstrating graph algorithms. You can

- Add a vertex by clicking the primary button in an open area and
- Remove a vertex by clicking at the vertex using the secondary button.
- Add an edge between two vertices by dragging from one vertex to the other vertex.
- Move a vertex by dragging the vertex while pressing the CTRL button pressed.

Figure 28.4 You can use the tool to create a graph with mouse gestures and show DFS/BFS trees, shortest paths, and other operations.

vertex type

Note

The vertices can be objects of any type. For example, you can consider cities as objects that contain the information such as its name, population, and mayor. Thus, you may define vertices as follows:

```
City city0 = new City("Seattle", 608660, "Mike McGinn");
...
City city11 = new City("Houston", 2099451, "Annise Parker");
City[] vertices = {city0, city1, . . . , city11};

public class City {
  private String cityName;
  private int population;
  private String mayor;

  public City(String cityName, int population, String mayor) {
    this.cityName = cityName;
    this.population = population;
    this.mayor = mayor;
  }
  public String getCityName() {
    return cityName;
  }

  public int getPopulation() {
    return population;
  }

  public String getMayor() {
    return mayor;
  }

  public void setMayor(String mayor) {
    this.mayor = mayor;
  }

  public void setPopulation(int population) {
    this.population = population;
  }
}
```

The vertices can be conveniently labeled using natural numbers 0, 1, 2, . . . , $n - 1$, for a graph of n vertices. Thus, `vertices[0]` represents `"Seattle"`, `vertices[1]` represents `"San Francisco"`, and so on, as shown in Figure 28.5.

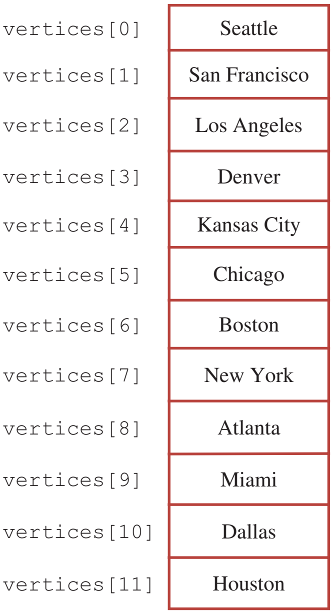

vertices[0]
Seattle
vertices[1]
San Francisco
vertices[2]
Los Angeles
vertices[3]
Denver
vertices[4]
Kansas City
vertices[5]
Chicago
vertices[6]
Boston
vertices[7]
New York
vertices[8]
Atlanta
vertices[9]
Miami
vertices[10]
Dallas
vertices[11]
Houston

FIGURE 28.5 An array stores the vertex names.

reference vertex

Note

You can reference a vertex by its name or its index, whichever is more convenient. Obviously, it is easy to access a vertex via its index in a program.

28.3.2 Representing Edges: Edge Array

The edges can be represented using a two-dimensional array. For example, you can store all the edges in the graph in Figure 28.1 using the following array:

```
int[][] edges = {
  {0, 1}, {0, 3}, {0, 5},
  {1, 0}, {1, 2}, {1, 3},
  {2, 1}, {2, 3}, {2, 4}, {2, 10},
  {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
  {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
  {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
  {6, 5}, {6, 7},
  {7, 4}, {7, 5}, {7, 6}, {7, 8},
  {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
  {9, 8}, {9, 11},
  {10, 2}, {10, 4}, {10, 8}, {10, 11},
  {11, 8}, {11, 9}, {11, 10}
};
```

This representation is known as the *edge array*. The vertices and edges in Figure 28.3a can be represented as follows:

edge array

```
String[] vertices = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};

int[][] edges = {{0, 2}, {1, 2}, {2, 4}, {3, 4}};
```

28.3.3 Representing Edges: Edge Objects

Another way to represent the edges is to define edges as objects and store the edges in a `java.util.ArrayList`. The `Edge` class can be defined as in Listing 28.1.

LISTING 28.1 `Edge.java`

```
public class Edge {
  int u;
  int v;

  public Edge(int u, int v) {
    this.u = u;
    this.v = v;
  }

  public boolean equals(Object o) {
    return u == ((Edge)o).u && v == ((Edge)o).v;
  }
}
```

For example, you can store all the edges in the graph in Figure 28.1 using the following list:

```
java.util.ArrayList<Edge> list = new java.util.ArrayList<>();
list.add(new Edge(0, 1));
list.add(new Edge(0, 3));
list.add(new Edge(0, 5));
...
```

Storing `Edge` objects in an `ArrayList` is useful if you don't know the edges in advance.

While representing edges using an edge array or `Edge` objects may be intuitive for input, it's not efficient for internal processing. The next two sections introduce the representation of graphs using *adjacency matrices* and *adjacency lists*. These two data structures are efficient for processing graphs.

28.3.4 Representing Edges: Adjacency Matrices

adjacency matrix

Assume the graph has *n* vertices. You can use a two-dimensional *n* × *n* matrix, say **`adjacencyMatrix`**, to represent the edges. Each element in the array is **`0`** or **`1`**. **`adjacencyMatrix[i][j]`** is **`1`** if there is an edge from vertex *i* to vertex *j*; otherwise, **`adjacencyMatrix[i][j]`** is **`0`**. If the graph is undirected, the matrix is symmetric, because **`adjacencyMatrix[i][j]`** is the same as **`adjacencyMatrix[j][i]`**. For example, the edges in the graph in Figure 28.1 can be represented using an *adjacency matrix* as follows:

```
int[][] adjacencyMatrix = {
  {0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0}, // Seattle
  {1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // San Francisco
  {0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0}, // Los Angeles
  {1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0}, // Denver
  {0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0}, // Kansas City
  {1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0}, // Chicago
  {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0}, // Boston
  {0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0}, // New York
  {0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1}, // Atlanta
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1}, // Miami
  {0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1}, // Dallas
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0}  // Houston
};
```

ragged array

> **Note**
> Since the matrix is symmetric for an undirected graph, to save storage you can use a ragged array.

The adjacency matrix for the directed graph in Figure 28.3a can be represented as follows:

```
int[][] a = {{0, 0, 1, 0, 0}, // Peter
             {0, 0, 1, 0, 0}, // Jane
             {0, 0, 0, 0, 1}, // Mark
             {0, 0, 0, 0, 1}, // Cindy
             {0, 0, 0, 0, 0}  // Wendy
            };
```

28.3.5 Representing Edges: Adjacency Lists

adjacency vertex lists
adjacency edge lists

You can represent edges using *adjacency vertex lists* or *adjacency edge lists*. An adjacency vertex list for a vertex *i* contains the vertices that are adjacent to *i* and an adjacency edge list for a vertex *i* contains the edges that are adjacent to *i*. You may define an array of lists. The array has *n* entries, and each entry is a list. The adjacency vertex list for vertex *i* contains all the vertices *j* such that there is an edge from vertex *i* to vertex *j*. For example, to represent the edges in the graph in Figure 28.1, you can create an array of lists as follows:

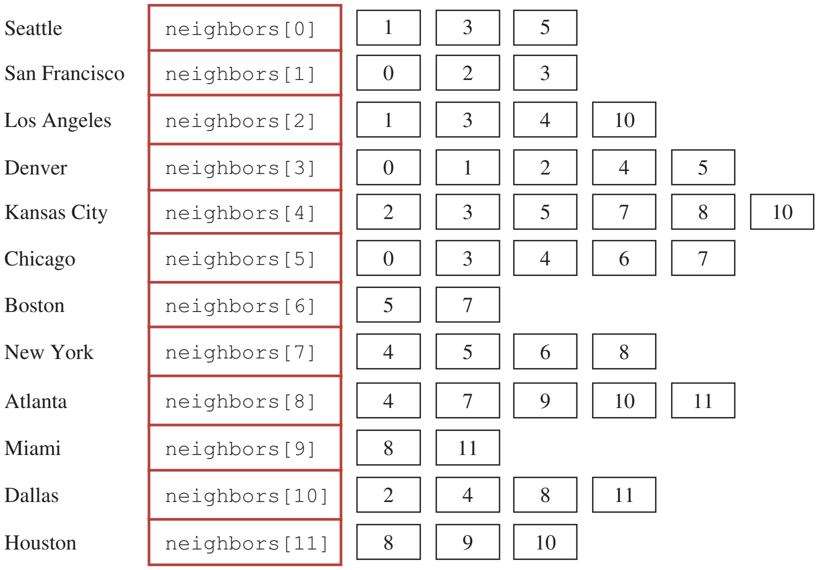
```
java.util.List<Integer>[] neighbors = new java.util.List[12];
```

`neighbors[0]` contains all vertices adjacent to vertex **`0`** (i.e., Seattle), **`neighbors[1]`** contains all vertices adjacent to vertex **`1`** (i.e., San Francisco), and so on, as shown in Figure 28.6.

To represent the adjacency edge lists for the graph in Figure 28.1, you can create an array of lists as follows:

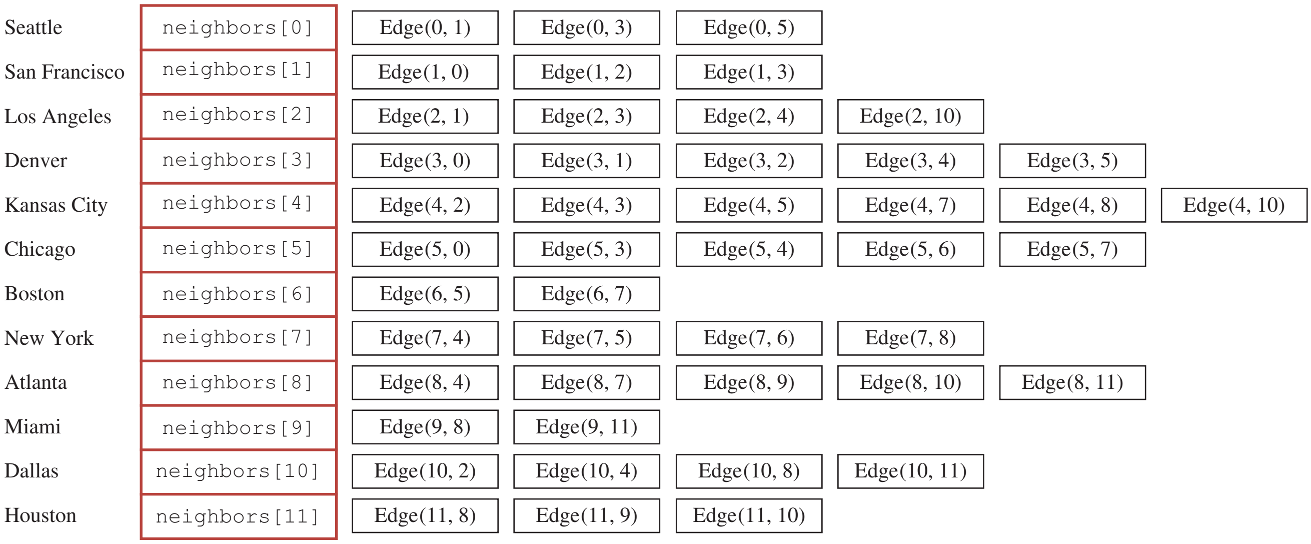
```
java.util.List<Edge>[] neighbors = new java.util.List[12];
```

`neighbors[0]` contains all edges adjacent to vertex **`0`** (i.e., Seattle), **`neighbors[1]`** contains all edges adjacent to vertex **`1`** (i.e., San Francisco), and so on, as shown in Figure 28.7.



Seattle	`neighbors[0]`	1	3	5			
San Francisco	`neighbors[1]`	0	2	3			
Los Angeles	`neighbors[2]`	1	3	4	10		
Denver	`neighbors[3]`	0	1	2	4	5	
Kansas City	`neighbors[4]`	2	3	5	7	8	10
Chicago	`neighbors[5]`	0	3	4	6	7	
Boston	`neighbors[6]`	5	7				
New York	`neighbors[7]`	4	5	6	8		
Atlanta	`neighbors[8]`	4	7	9	10	11	
Miami	`neighbors[9]`	8	11				
Dallas	`neighbors[10]`	2	4	8	11		
Houston	`neighbors[11]`	8	9	10			

FIGURE 28.6 Edges in the graph in Figure 28.1 are represented using adjacency vertex lists.



Seattle	`neighbors[0]`	Edge(0, 1)	Edge(0, 3)	Edge(0, 5)			
San Francisco	`neighbors[1]`	Edge(1, 0)	Edge(1, 2)	Edge(1, 3)			
Los Angeles	`neighbors[2]`	Edge(2, 1)	Edge(2, 3)	Edge(2, 4)	Edge(2, 10)		
Denver	`neighbors[3]`	Edge(3, 0)	Edge(3, 1)	Edge(3, 2)	Edge(3, 4)	Edge(3, 5)	
Kansas City	`neighbors[4]`	Edge(4, 2)	Edge(4, 3)	Edge(4, 5)	Edge(4, 7)	Edge(4, 8)	Edge(4, 10)
Chicago	`neighbors[5]`	Edge(5, 0)	Edge(5, 3)	Edge(5, 4)	Edge(5, 6)	Edge(5, 7)	
Boston	`neighbors[6]`	Edge(6, 5)	Edge(6, 7)				
New York	`neighbors[7]`	Edge(7, 4)	Edge(7, 5)	Edge(7, 6)	Edge(7, 8)		
Atlanta	`neighbors[8]`	Edge(8, 4)	Edge(8, 7)	Edge(8, 9)	Edge(8, 10)	Edge(8, 11)	
Miami	`neighbors[9]`	Edge(9, 8)	Edge(9, 11)				
Dallas	`neighbors[10]`	Edge(10, 2)	Edge(10, 4)	Edge(10, 8)	Edge(10, 11)		
Houston	`neighbors[11]`	Edge(11, 8)	Edge(11, 9)	Edge(11, 10)			

FIGURE 28.7 Edges in the graph in Figure 28.1 are represented using adjacency edge lists.

adjacency matrices vs.
adjacency lists

Note

You can represent a graph using an adjacency matrix or adjacency lists. Which one is better? If the graph is dense (i.e., there are a lot of edges), using an adjacency matrix is preferred. If the graph is very sparse (i.e., very few edges), using adjacency lists is better, because using an adjacency matrix would waste a lot of space.

Both adjacency matrices and adjacency lists can be used in a program to make algorithms more efficient. For example, it takes $O(1)$ constant time to check whether two vertices are connected using an adjacency matrix, and it takes linear time $O(m)$ to print all edges in a graph using adjacency lists, where m is the number of edges.

adjacency vertex lists vs. adjacency edge lists

Note

Adjacency vertex list is simpler for representing unweighted graphs. However, adjacency edge lists are more flexible for a wide range of graph applications. It is easy to add additional constraints on edges using adjacency edge lists. For this reason, this book will use adjacency edge lists to represent graphs.

using `ArrayList`

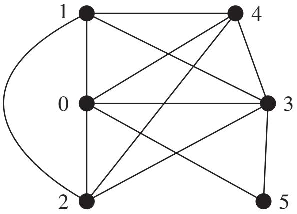
You can use arrays, array lists, or linked lists to store adjacency lists. We will use lists instead of arrays, because the lists are easily expandable to enable you to add new vertices. Further we will use array lists instead of linked lists, because our algorithms only require searching for adjacent vertices in the list. Using array lists is more efficient for our algorithms. Using array lists, the adjacency edge list in Figure 28.6 can be built as follows:

```
List<ArrayList<Edge>> neighbors = new ArrayList<>();
neighbors.add(new ArrayList<Edge>());
neighbors.get(0).add(new Edge(0, 1));
neighbors.get(0).add(new Edge(0, 3));
neighbors.get(0).add(new Edge(0, 5));
neighbors.add(new ArrayList<Edge>());
neighbors.get(1).add(new Edge(1, 0));
neighbors.get(1).add(new Edge(1, 2));
neighbors.get(1).add(new Edge(1, 3));
...
...
neighbors.get(11).add(new Edge(11, 8));
neighbors.get(11).add(new Edge(11, 9));
neighbors.get(11).add(new Edge(11, 10));
```

Check Point

28.3.1 How do you represent vertices in a graph? How do you represent edges using an edge array? How do you represent an edge using an edge object? How do you represent edges using an adjacency matrix? How do you represent edges using adjacency lists?

28.3.2 Represent the following graph using an edge array, a list of edge objects, an adjacency matrix, an adjacency vertex list, and an adjacency edge list, respectively.



28.4 Modeling Graphs

Key Point

The `Graph` interface defines the common operations for a graph.

The Java Collections Framework serves as a good example for designing complex data structures. The common features of data structures are defined in the interfaces (e.g., `Collection`, `Set`, `List`, `Queue`), as shown in Figure 20.1. This design pattern is useful for modeling graphs. We will define an interface named `Graph` that contains all the common operations of graphs. Many concrete graphs can be added to the design. For example, we will define such graphs named `UnweightedGraph` and `WeightedGraph`. The relationships of these interfaces and classes are illustrated in Figure 28.8.

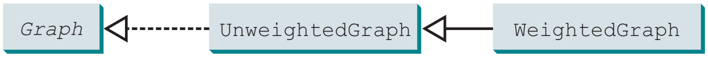


FIGURE 28.8 Common operations of graphs are defined in the interface and concrete classes define concrete graphs.

What are the common operations for a graph? In general, you need to get the number of vertices in a graph, get all vertices in a graph, get the vertex object with a specified index, get the index of the vertex with a specified name, get the neighbors for a vertex, get the degree for a vertex, clear the graph, add a new vertex, add a new edge, perform a depth-first search, and perform a breadth-first search. Depth-first search and breadth-first search will be introduced in the next section. Figure 28.9 illustrates these methods in the UML diagram.

`UnweightedGraph` does not introduce any new methods. A list of vertices and an edge adjacency list are defined in the class. With these data fields, it is sufficient to implement all the methods defined in the `Graph` interface. For convenience, we assume the graph is a simple graph, that is, a vertex has no edge to itself and there are no parallel edges from vertex *u* to *v*.

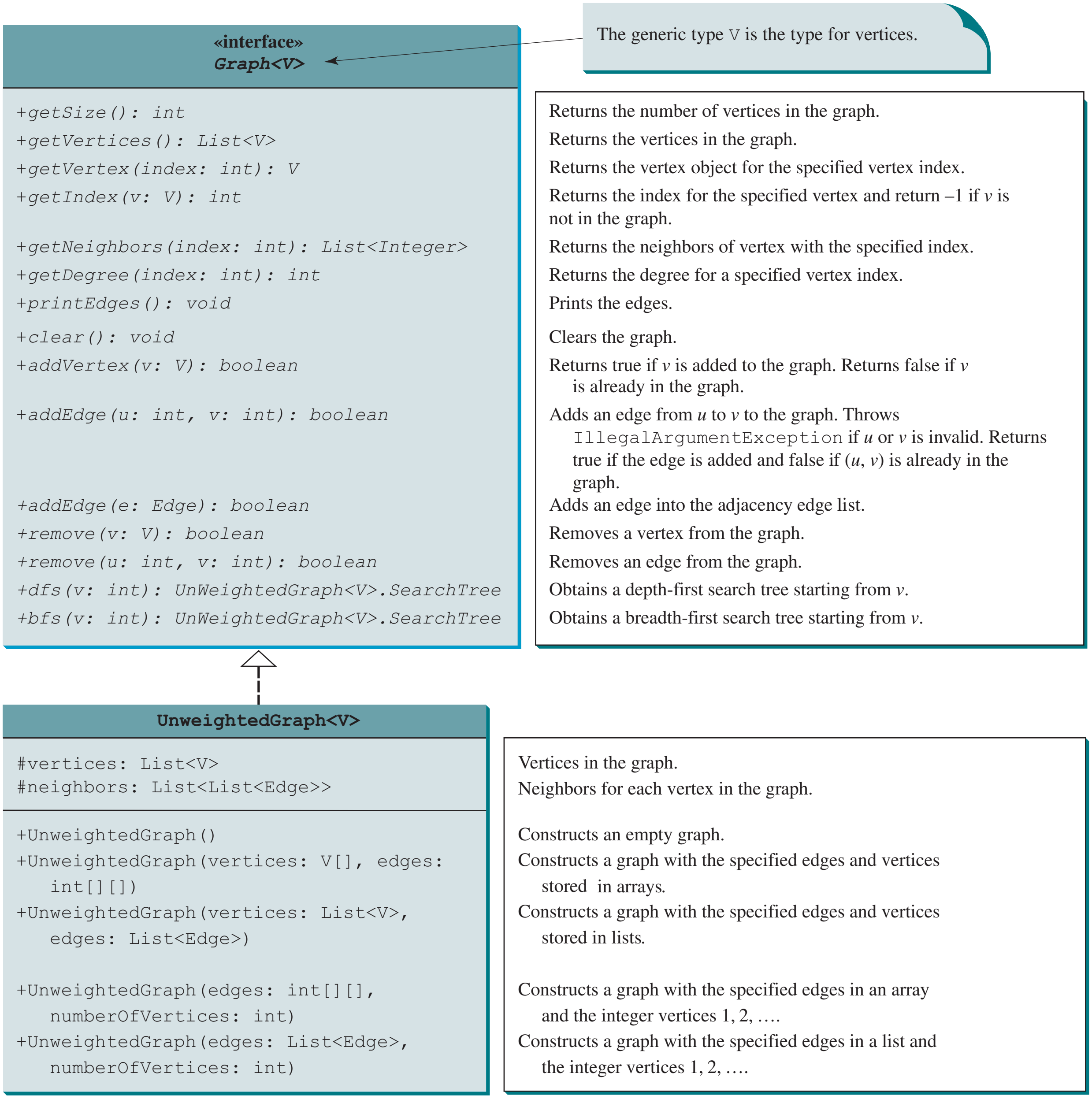

«interface»
Graph<V>
The generic type V is the type for vertices.
+getSize(): int
+getVertices(): List<V>
+getVertex(index: int): V
+getIndex(v: V): int
+getNeighbors(index: int): List<Integer>
+getDegree(index: int): int
+printEdges(): void
+clear(): void
+addVertex(v: V): boolean
+addEdge(u: int, v: int): boolean
+addEdge(e: Edge): boolean
+remove(v: V): boolean
+remove(u: int, v: int): boolean
+dfs(v: int): UnWeightedGraph<V>.SearchTree
+bfs(v: int): UnWeightedGraph<V>.SearchTree
Returns the number of vertices in the graph.
Returns the vertices in the graph.
Returns the vertex object for the specified vertex index.
Returns the index for the specified vertex and return –1 if v is not in the graph.
Returns the neighbors of vertex with the specified index.
Returns the degree for a specified vertex index.
Prints the edges.
Clears the graph.
Returns true if v is added to the graph. Returns false if v is already in the graph.
Adds an edge from u to v to the graph. Throws IllegalArgumentException if u or v is invalid. Returns true if the edge is added and false if (u, v) is already in the graph.
Adds an edge into the adjacency edge list.
Removes a vertex from the graph.
Removes an edge from the graph.
Obtains a depth-first search tree starting from v.
Obtains a breadth-first search tree starting from v.
UnweightedGraph<V>
#vertices: List<V>
#neighbors: List<List<Edge>>
+UnweightedGraph()
+UnweightedGraph(vertices: V[], edges: int[][])
+UnweightedGraph(vertices: List<V>, edges: List<Edge>)
+UnweightedGraph(edges: int[][], numberOfVertices: int)
+UnweightedGraph(edges: List<Edge>, numberOfVertices: int)
Vertices in the graph.
Neighbors for each vertex in the graph.
Constructs an empty graph.
Constructs a graph with the specified edges and vertices stored in arrays.
Constructs a graph with the specified edges and vertices stored in lists.
Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ….
Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ….

FIGURE 28.9 The `Graph` interface defines the common operations for all types of graphs.

vertices and their indices

Note

You can create a graph with any type of vertices. Each vertex is associated with an index, which is the same as the index of the vertex in the vertices list. If you create a graph without specifying the vertices, the vertices are the same as their indices.

Assume the **`Graph`** interface and the **`UnweightedGraph`** class are available. Listing 28.2 gives a test program that creates the graph in Figure 28.1 and another graph for the one in Figure 28.3a.

Listing 28.2 TestGraph.java

vertices

edges

create a graph

number of vertices

get vertex

get index

print edges

list of Edge objects

create a graph

print edges

```java
public class TestGraph {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    // Edge array for graph in Figure 28.1
    int[][] edges = {
      {0, 1}, {0, 3}, {0, 5},
      {1, 0}, {1, 2}, {1, 3},
      {2, 1}, {2, 3}, {2, 4}, {2, 10},
      {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
      {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
      {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
      {6, 5}, {6, 7},
      {7, 4}, {7, 5}, {7, 6}, {7, 8},
      {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
      {9, 8}, {9, 11},
      {10, 2}, {10, 4}, {10, 8}, {10, 11},
      {11, 8}, {11, 9}, {11, 10}
    };

    Graph<String> graph1 = new UnweightedGraph<>(vertices, edges);
    System.out.println("The number of vertices in graph1: "
      + graph1.getSize());
    System.out.println("The vertex with index 1 is "
      + graph1.getVertex(1));
    System.out.println("The index for Miami is " +
      graph1.getIndex("Miami"));
    System.out.println("The edges for graph1:");
    graph1.printEdges();

    // List of Edge objects for graph in Figure 28.3a
    String[] names = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};
    java.util.ArrayList<Edge> edgeList
      = new java.util.ArrayList<>();
    edgeList.add(new Edge(0, 2));
    edgeList.add(new Edge(1, 2));
    edgeList.add(new Edge(2, 4));
    edgeList.add(new Edge(3, 4));
    // Create a graph with 5 vertices
    Graph<String> graph2 = new UnweightedGraph<>
      (java.util.Arrays.asList(names), edgeList);
    System.out.println("\nThe number of vertices in graph2: "
      + graph2.getSize());
    System.out.println("The edges for graph2:");
    graph2.printEdges();
  }
}
```

```
The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Seattle (0): (0, 1) (0, 3) (0, 5)
San Francisco (1): (1, 0) (1, 2) (1, 3)
Los Angeles (2): (2, 1) (2, 3) (2, 4) (2, 10)
Denver (3): (3, 0) (3, 1) (3, 2) (3, 4) (3, 5)
Kansas City (4): (4, 2) (4, 3) (4, 5) (4, 7) (4, 8) (4, 10)
Chicago (5): (5, 0) (5, 3) (5, 4) (5, 6) (5, 7)
Boston (6): (6, 5) (6, 7)
New York (7): (7, 4) (7, 5) (7, 6) (7, 8)
Atlanta (8): (8, 4) (8, 7) (8, 9) (8, 10) (8, 11)
Miami (9): (9, 8) (9, 11)
Dallas (10): (10, 2) (10, 4) (10, 8) (10, 11)
Houston (11): (11, 8) (11, 9) (11, 10)

The number of vertices in graph2: 5
The edges for graph2:
Peter (0): (0, 2)
Jane (1): (1, 2)
Mark (2): (2, 4)
Cindy (3): (3, 4)
Wendy (4):
```

The program creates **`graph1`** for the graph in Figure 28.1 in lines 3–23. The vertices for **`graph1`** are defined in lines 3–5. The edges for **`graph1`** are defined in 8–21. The edges are represented using a two-dimensional array. For each row **`i`** in the array, **`edges[i][0]`** and **`edges[i][1]`** indicate there is an edge from vertex **`edges[i][0]`** to vertex **`edges[i][1]`**. For example, the first row, {**`0`**, **`1`**}, represents the edge from vertex **`0`** (**`edges[0][0]`**) to vertex **`1`** (**`edges[0][1]`**). The row {**`0`**, **`5`**} represents the edge from vertex **`0`** (**`edges[2][0]`**) to vertex **`5`** (**`edges[2][1]`**). The graph is created in line 23. Line 31 invokes the **`printEdges()`** method on **`graph1`** to display all edges in **`graph1`**.

The program creates **`graph2`** for the graph in Figure 28.3a in lines 34–43. The edges for **`graph2`** are defined in lines 37–40. **`graph2`** is created using a list of **`Edge`** objects in line 43. Line 47 invokes the **`printEdges()`** method on **`graph2`** to display all edges in **`graph2`**.

Note both **`graph1`** and **`graph2`** contain the vertices of strings. The vertices are associated with indices **`0`**, **`1`**, . . . , **`n-1`**. The index is the location of the vertex in **`vertices`**. For example, the index of vertex **`Miami`** is **`9`**.

Now, we turn our attention to implementing the interface and classes. Listings 28.3 and Listings 28.4 give the **`Graph`** interface and the **`UnweightedGraph`** class, respectively.

LISTING 28.3 Graph.java

```java
public interface Graph<V> {
  /** Return the number of vertices in the graph */
  public int getSize();

  /** Return the vertices in the graph */
  public java.util.List<V> getVertices();

  /** Return the object for the specified vertex index */
  public V getVertex(int index);

  /** Return the index for the specified vertex object */
```

getSize

getVertices

getVertex

getIndex
getNeighbors
getDegree
printEdges
clear
addVertex
addEdge
addEdge
remove vertex
remove edge
dfs
bfs

```
  public int getIndex(V v);

  /** Return the neighbors of vertex with the specified index */
  public java.util.List<Integer> getNeighbors(int index);

  /** Return the degree for a specified vertex */
  public int getDegree(int v);

  /** Print the edges */
  public void printEdges();

  /** Clear the graph */
  public void clear();

  /** Add a vertex to the graph */
  public boolean addVertex(V vertex);

  /** Add an edge to the graph */
  public boolean addEdge(int u, int v);

  /** Add an edge to the graph */
  public boolean addEdge(Edge e);

  /** Remove a vertex v from the graph, return true if successful */
  public boolean remove(V v);

  /** Remove an edge (u, v) from the graph, return true if successful */
  public boolean remove(int u, int v);

  /** Obtain a depth-first search tree */
  public UnweightedGraph<V>.SearchTree dfs(int v);

  /** Obtain a breadth-first search tree */
  public UnweightedGraph<V>.SearchTree bfs(int v);
}
```

Listing 28.4 UnweightedGraph.java

no-arg constructor
constructor
constructor

```
import java.util.*;

public class UnweightedGraph<V> implements Graph<V> {
  protected List<V> vertices = new ArrayList<>(); // Store vertices
  protected List<List<Edge>> neighbors
    = new ArrayList<>(); // Adjacency Edge lists

  /** Construct an empty graph */
  protected UnweightedGraph() {
  }

  /** Construct a graph from vertices and edges stored in arrays */
  protected UnweightedGraph(V[] vertices, int[][] edges) {
    for (int i = 0; i < vertices.length; i++)
      addVertex(vertices[i]);

    createAdjacencyLists(edges, vertices.length);
  }

  /** Construct a graph from vertices and edges stored in List */
  protected UnweightedGraph(List<V> vertices, List<Edge> edges) {
    for (int i = 0; i < vertices.size(); i++)
```

```
      addVertex(vertices.get(i));

    createAdjacencyLists(edges, vertices.size());
  }

```

constructor

```
  /** Construct a graph for integer vertices 0, 1, 2 and edge list */
  protected UnweightedGraph(List<Edge> edges, int numberOfVertices) {
    for (int i = 0; i < numberOfVertices; i++)
      addVertex((V)(Integer.valueOf(i))); // vertices is {0, 1, . . . }

    createAdjacencyLists(edges, numberOfVertices);
  }

```

constructor

```
  /** Construct a graph from integer vertices 0, 1, and edge array */
  protected UnweightedGraph(int[][] edges, int numberOfVertices) {
    for (int i = 0; i < numberOfVertices; i++)
      addVertex((V)(Integer.valueOf(i))); // vertices is {0, 1, . . . }

    createAdjacencyLists(edges, numberOfVertices);
  }

  /** Create adjacency lists for each vertex */
  private void createAdjacencyLists(
      int[][] edges, int numberOfVertices) {
    for (int i = 0; i < edges.length; i++) {
      addEdge(edges[i][0], edges[i][1]);
    }
  }

  /** Create adjacency lists for each vertex */
  private void createAdjacencyLists(
      List<Edge> edges, int numberOfVertices) {
    for (Edge edge: edges) {
      addEdge(edge.u, edge.v);
    }
  }

```

`getSize`

```
  @Override /** Return the number of vertices in the graph */
  public int getSize() {
    return vertices.size();
  }

```

`getVertices`

```
  @Override /** Return the vertices in the graph */
  public List<V> getVertices() {
    return vertices;
  }

```

`getVertex`

```
  @Override /** Return the object for the specified vertex */
  public V getVertex(int index) {
    return vertices.get(index);
  }

```

`getIndex`

```
  @Override /** Return the index for the specified vertex object */
  public int getIndex(V v) {
    return vertices.indexOf(v);
  }

```

`getNeighbors`

```
  @Override /** Return the neighbors of the specified vertex */
  public List<Integer> getNeighbors(int index) {
    List<Integer> result = new ArrayList<>();
    for (Edge e: neighbors.get(index))
      result.add(e.v);

    return result;
  }

```

getDegree

```
  @Override /** Return the degree for a specified vertex */
  public int getDegree(int v) {
    return neighbors.get(v).size();
  }

```

printEdges

```
  @Override /** Print the edges */
  public void printEdges() {
    for (int u = 0; u < neighbors.size(); u++) {
      System.out.print(getVertex(u) + " (" + u + "): ");
      for (Edge e: neighbors.get(u)) {
        System.out.print("(" + getVertex(e.u) + ", " +
          getVertex(e.v) + ") ");
      }
      System.out.println();
    }
  }

```

clear

```
  @Override /** Clear the graph */
  public void clear() {
    vertices.clear();
    neighbors.clear();
  }

```

addVertex

```
  @Override /** Add a vertex to the graph */
  public boolean addVertex(V vertex) {
    if (!vertices.contains(vertex)) {
      vertices.add(vertex);
      neighbors.add(new ArrayList<Edge>());
      return true;
    }
    else {
      return false;
    }
  }

```

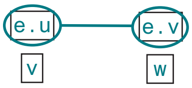
addEdge

```
  @Override /** Add an edge to the graph */
  public boolean addEdge(Edge e) {
    if (e.u < 0 || e.u > getSize() - 1)
      throw new IllegalArgumentException("No such index: " + e.u);

    if (e.v < 0 || e.v > getSize() -  1)
      throw new IllegalArgumentException("No such index: " + e.v);

    if (!neighbors.get(e.u).contains(e)) {
      neighbors.get(e.u).add(e);
      return true;
    }
    else {
      return false;
    }
  }

```

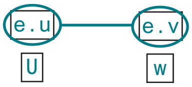
addEdge overloaded

```
  @Override /** Add an edge to the graph */
  public boolean addEdge(int u, int v) {
    return addEdge(new Edge(u, v));
  }

  @Override /** Obtain a DFS tree starting from vertex v */
  /** To be discussed in Section 28.7 */
  public SearchTree dfs(int v) {
    List<Integer> searchOrder = new ArrayList<>();
    int[] parent = new int[vertices.size()];
    for (int i = 0; i < parent.length; i++)
      parent[i] = -1; // Initialize parent[i] to -1

    // Mark visited vertices
    boolean[] isVisited = new boolean[vertices.size()];

    // Recursively search
    dfs(v, parent, searchOrder, isVisited);

    // Return a search tree
    return new SearchTree(v, parent, searchOrder);
  }

  /** Recursive method for DFS search */
  private void dfs(int v, int[] parent, List<Integer> searchOrder,
      boolean[] isVisited) {
    // Store the visited vertex
    searchOrder.add(v);
    isVisited[v] = true; // Vertex v visited

    for (Edge e : neighbors.get(v)) {// e.u is v
      int w = e.v; // e.v is w in Listing 28.8
      if (!isVisited[w]) {
        parent[w] = v; // The parent of vertex w is v
        dfs(w, parent, searchOrder, isVisited); // Recursive search
      }
    }
  }

  @Override /** Starting bfs search from vertex v */
  /** To be discussed in Section 28.9 */
  public SearchTree bfs(int v) {
    List<Integer> searchOrder = new ArrayList<>();
    int[] parent = new int[vertices.size()];
    for (int i = 0; i < parent.length; i++)
      parent[i] = -1; // Initialize parent[i] to -1

    java.util.LinkedList<Integer> queue =
      new java.util.LinkedList<>(); // list used as a queue
    boolean[] isVisited = new boolean[vertices.size()];
    queue.offer(v); // Enqueue v
    isVisited[v] = true; // Mark it visited

    while (!queue.isEmpty()) {
      int u = queue.poll(); // Dequeue to u
      searchOrder.add(u); // u searched
      for (Edge e: neighbors.get(u)) {// Note that e.u is u
        int w = e.v; // e.v is w in Listing 28.8
        if (!isVisited[w]) {// e.v is w in Listing 28.11
          queue.offer(w); // Enqueue w
          parent[w] = u; // The parent of w is u
          isVisited[w] = true; // Mark it visited
        }
      }
```

dfs method



bfs method



SearchTree inner class

```
    }

    return new SearchTree(v, parent, searchOrder);
  }

  /** Tree inner class inside the UnweightedGraph class */
  /** To be discussed in Section 28.6 */
  public class SearchTree {
    private int root; // The root of the tree
    private int[] parent; // Store the parent of each vertex
    private List<Integer> searchOrder; // Store the search order

    /** Construct a tree with root, parent, and searchOrder */
    public SearchTree(int root, int[] parent,
        List<Integer> searchOrder) {
      this.root = root;
      this.parent = parent;
      this.searchOrder = searchOrder;
    }

    /** Return the root of the tree */
    public int getRoot() {
      return root;
    }

    /** Return the parent of vertex v */
    public int getParent(int v) {
      return parent[v];
    }

    /** Return an array representing search order */
    public List<Integer> getSearchOrder() {
      return searchOrder;
    }

    /** Return number of vertices found */
    public int getNumberOfVerticesFound() {
      return searchOrder.size();
    }

    /** Return the path of vertices from a vertex to the root */
    public List<V> getPath(int index) {
      ArrayList<V> path = new ArrayList<>();

      do {
        path.add(vertices.get(index));
        index = parent[index];
      }
      while (index != -1);

      return path;
    }

    /** Print a path from the root to vertex v */
    public void printPath(int index) {
      List<V> path = getPath(index);
      System.out.print("A path from " + vertices.get(root) + " to " +
        vertices.get(index) + ": ");
      for (int i = path.size() - 1; i >= 0; i--)
        System.out.print(path.get(i) + " ");
    }
```

```
266
267      /** Print the whole tree */
268      public void printTree() {
269        System.out.println("Root is: " + vertices.get(root));
270        System.out.print("Edges: ");
271        for (int i = 0; i < parent.length; i++) {
272          if (parent[i] != -1) {
273            // Display an edge
274            System.out.print("(" + vertices.get(parent[i]) + ", " +
275              vertices.get(i) + ") ");
276          }
277        }
278        System.out.println();
279      }
280    }
281
282    @Override /** Remove vertex v and return true if successful */
283    public boolean remove(V v) {
284      return true; // Implementation left as an exercise
285    }
286
287    @Override /** Remove edge (u, v) and return true if successful */
288    public boolean remove(int u, int v) {
289      return true; // Implementation left as an exercise
290    }
291  }
```

The code in the `Graph` interface in Listing 28.3 is straightforward. Let us digest the code in the `UnweightedGraph` class in Listing 28.4.

The `UnweightedGraph` class defines the data field `vertices` (line 4) to store vertices and `neighbors` (line 5) to store edges in adjacency `edges` lists. `neighbors.get(i)` stores all edges adjacent to vertex `i`. Four overloaded constructors are defined in lines 9–42 to create a default graph, or a graph from arrays or lists of edges and vertices. The `createAdjacencyLists(int[][] edges, int numberOfVertices)` method creates adjacency lists from edges in an array (lines 45–50). The `createAdjacencyLists (List<Edge> edges, int numberOfVertices)` method creates adjacency lists from edges in a list (lines 53–58).

The `getNeighbors(u)` method (lines 81–87) returns a list of vertices adjacent to vertex `u`. The `clear()` method (lines 106–110) removes all vertices and edges from the graph. The `addVertex(u)` method (lines 112–122) adds a new vertex to `vertices` and returns true. It returns false if the vertex is already in the graph (line 120).

The `addEdge(e)` method (lines 124–139) adds a new edge to the adjacency edge list and returns true. It returns false if the edge is already in the graph. This method may throw `IllegalArgumentExcepiton` if the edge is invalid (lines 126–130).

The `addEdge(u, v)` method (lines 141–144) adds an edge (`u`, `v`) to the graph. If a graph is undirected, you should invoke `addEdge(u, v)` and `addEdge(v, u)` to add an edge between `u` and `v`.

The `printEdges()` method (lines 95–104) displays all vertices and edges adjacent to each vertex.

The code in lines 148–208 gives the methods for finding a depth-first search tree and a breadth-first search tree, which will be introduced in Sections 28.7 and 28.9, respectively.

Check Point

28.4.1 Describe the methods in `Graph` and `UnweightedGraph`.

28.4.2 For the code in Listing 28.2, TestGraph.java, what is `graph1.getIndex("Seattle")`? What is `graph1.getDegree(5)`? What is `graph1.getVertex(4)`?

28.4.3 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    Graph<Character> graph = new UnweightedGraph<>();
    graph.addVertex('U');
    graph.addVertex('V');
    int indexForU = graph.getIndex('U');
    int indexForV = graph.getIndex('V');
    System.out.println("indexForU is " + indexForU);
    System.out.println("indexForV is " + indexForV);
    graph.addEdge(indexForU, indexForV);
    System.out.println("Degree of U is " +
      graph.getDegree(indexForU));
    System.out.println("Degree of V is " +
      graph.getDegree(indexForV));
  }
}
```

28.4.4 What will `getIndex(v)` return if v is not in the graph? What happens to `getVertex(index)` if index is not in the graph? What happens to `addVertex(v)` if v is already in the graph? What happens to `addEdge(u, v)` if u or v is not in the graph?

28.5 Graph Visualization

Key Point

To display a graph visually, each vertex must be assigned a location.

The preceding section introduced the `Graph` interface and the `UnweightedGraph` class. This section discusses how to display graphs graphically. In order to display a graph, you need to know where each vertex is displayed and the name of each vertex. To ensure a graph can be displayed, we define an interface named `Displayable` that has the methods for obtaining the `x-` and `y-`coordinates and their names, and make vertices instances of `Displayable`, in Listing 28.5.

Listing 28.5 Displayable.java

`Displayable` interface

```
public interface Displayable {
  public double getX(); // Get x-coordinate of the vertex
  public double getY(); // Get y-coordinate of the vertex
  public String getName(); // Get display name of the vertex
}
```

A graph with `Displayable` vertices can now be displayed on a pane named `GraphView`, as shown in Listing 28.6.

Listing 28.6 GraphView.java

extends `BorderPane`

`Displayable` vertices

```
import javafx.scene.Group;
import javafx.scene.layout.BorderPane;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;

public class GraphView extends BorderPane {
  private Graph<? extends Displayable> graph;
  private Group group = new Group();

  public GraphView(Graph<? extends Displayable> graph) {
    this.graph = graph;
    this.setCenter(group); // Center the group
14       repaintGraph();
15     }
16
17     private void repaintGraph() {
18       group.getChildren().clear(); // Clear group for a new display
19
20       // Draw vertices and text for vertices
21       java.util.List<? extends Displayable> vertices
22         = graph.getVertices();
23       for (int i = 0; i < graph.getSize(); i++) {
24         double x = vertices.get(i).getX();
25         double y = vertices.get(i).getY();
26         String name = vertices.get(i).getName();
27
28         group.getChildren().add(new Circle(x, y, 16));
29         group.getChildren().add(new Text(x - 8, y - 18, name));
30       }
31
32       // Draw edges for pairs of vertices
33       for (int i = 0; i < graph.getSize(); i++) {
34         java.util.List<Integer> neighbors = graph.getNeighbors(i);
35         double x1 = graph.getVertex(i).getX();
36         double y1 = graph.getVertex(i).getY();
37         for (int v: neighbors) {
38           double x2 = graph.getVertex(v).getX();
39           double y2 = graph.getVertex(v).getY();
40
41           // Draw an edge for (i, v)
42           group.getChildren().add(new Line(x1, y1, x2, y2));
43         }
44       }
45     }
46   }
```

display a vertex

display a text

draw an edge

To display a graph on a pane, simply create an instance of **`GraphView`** by passing the graph as an argument in the constructor (line 11). The class for the graph's vertex must implement the **`Displayable`** interface in order to display the vertices (lines 21–44). For each vertex index **`i`**, invoking **`graph.getNeighbors(i)`** returns its adjacency list (line 34). From this list, you can find all vertices that are adjacent to **`i`** and draw a line to connect **`i`** with its adjacent vertex (lines 35–42).

Listing 28.7 gives an example of displaying the graph in Figure 28.1, as shown in Figure 28.10.

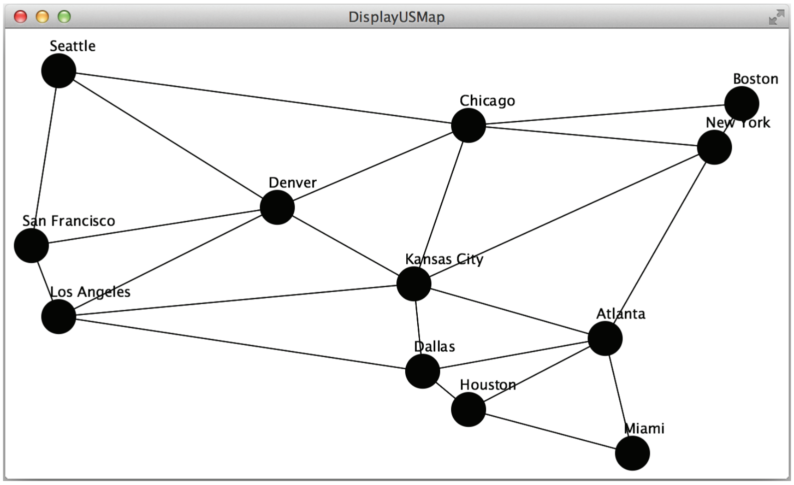


FIGURE 28.10 The graph is displayed in the pane.

Listing 28.7 DisplayUSMap.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class DisplayUSMap extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    City[] vertices = {new City("Seattle", 75, 50),
      new City("San Francisco", 50, 210),
      new City("Los Angeles", 75, 275), new City("Denver", 275, 175),
      new City("Kansas City", 400, 245),
      new City("Chicago", 450, 100), new City("Boston", 700, 80),
      new City("New York", 675, 120), new City("Atlanta", 575, 295),
      new City("Miami", 600, 400), new City("Dallas", 408, 325),
      new City("Houston", 450, 360) };

    // Edge array for graph in Figure 28.1
    int[][] edges = {
      {0, 1}, {0, 3}, {0, 5}, {1, 0}, {1, 2}, {1, 3},
      {2, 1}, {2, 3}, {2, 4}, {2, 10},
      {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
      {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
      {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
      {6, 5}, {6, 7}, {7, 4}, {7, 5}, {7, 6}, {7, 8},
      {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
      {9, 8}, {9, 11}, {10, 2}, {10, 4}, {10, 8}, {10, 11},
      {11, 8}, {11, 9}, {11, 10}
    };

    Graph<City> graph = new UnweightedGraph<>(vertices, edges);

    // Create a scene and place it in the stage
    Scene scene = new Scene(new GraphView(graph), 750, 450);
    primaryStage.setTitle("DisplayUSMap"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  static class City implements Displayable {
    private double x, y;
    private String name;

    City(String name, double x, double y) {
      this.name = name;
      this.x = x;
      this.y = y;
    }

    @Override
    public double getX() {
      return x;
    }

    @Override
    public double getY() {
      return y;
    }

```

create a graph

create a GraphView

City class

```
59      @Override
60      public String getName() {
61        return name;
62      }
63    }
64  }
```

The class `City` is defined to model the vertices with their coordinates and names (lines 39–63). The program creates a graph with the vertices of the `City` type (line 30). Since `City` implements `Displayable`, a `GraphView` object created for the graph displays the graph in the pane (line 33).

As an exercise to get acquainted with the graph classes and interfaces, add a city (e.g., Savannah) with appropriate edges into the graph.

Check Point

28.5.1 Will Listing 28.7, DisplayUSMap.java work, if the code in lines 38–42 in Listing 28.6, GraphView.java is replaced by the following code?

```
if (i < v) {
  double x2 = graph.getVertex(v).getX();
  double y2 = graph.getVertex(v).getY();

  // Draw an edge for (i, v)
  getChildren().add(new Line(x1, y1, x2, y2));
}
```

28.5.2 For the `graph1` object created in Listing 28.1, TestGraph.java, can you create a `GraphView` object as follows?

```
GraphView view = new GraphView(graph1);
```

28.6 Graph Traversals

Depth-first and breadth-first are two common ways to traverse a graph.

Key Point

depth-first search
breadth-first search

Graph traversal is the process of visiting each vertex in the graph exactly once. There are two popular ways to traverse a graph: *depth-first traversal* (or *depth-first search*) and *breadth-first traversal* (or *breadth-first search*). Both traversals result in a spanning tree, which can be modeled using a class, as shown in Figure 28.11. Note `SearchTree` is an inner class defined in the `UnweightedGraph` class. `UnweightedGraph<V>.SearchTree` is different from the `Tree` interface defined in Section 25.2.5. `UnweightedGraph<V>.SearchTree` is a specialized class designed for describing the parent–child relationship of the nodes, whereas the `Tree` interface defines common operations such as searching, inserting, and deleting in a tree. Since there is no need to perform these operations for a spanning tree, `UnweightedGraph<V>.SearchTree` is not defined as a subtype of `Tree`.

The `SearchTree` class is defined as an inner class in the `UnweightedGraph` class in lines 210–278 in Listing 28.4. The constructor creates a tree with the root, edges, and a search order.

The `SearchTree` class defines seven methods. The `getRoot()` method returns the root of the tree. You can get the order of the vertices searched by invoking the `getSearchOrder()` method. You can invoke `getParent(v)` to find the parent of vertex `v` in the search. Invoking `getNumberOfVerticesFound()` returns the number of vertices searched. The method `getPath(index)` returns a list of vertices from the specified vertex index to the root. Invoking `printPath(v)` displays a path from the root to `v`. You can display all edges in the tree using the `printTree()` method.

Sections 28.7 and 28.9 will introduce depth-first search and breadth-first search, respectively. Both searches will result in an instance of the `SearchTree` class.

Check Point

28.6.1 Does `UnweightedGraph<V>.SearchTree` implement the `Tree` interface defined in Listing 25.3, Tree.java?

28.6.2 What method do you use to find the parent of a vertex in the tree?

UnweightedGraph<V>.SearchTree	
`-root: int`	The root of the tree.
`-parent: int[]`	The parents of the vertices.
`-searchOrder: List<Integer>`	The orders for traversing the vertices.
`+SearchTree(root: int, parent: int[], searchOrder: List<Integer>)`	Constructs a tree with the specified root, parent, and `searchOrder`.
`+getRoot(): int`	Returns the root of the tree.
`+getSearchOrder(): List<Integer>`	Returns the order of vertices searched.
`+getParent(index: int): int`	Returns the parent for the specified vertex index.
`+getNumberOfVerticesFound(): int`	Returns the number of vertices searched.
`+getPath(index: int): List<V>`	Returns a list of vertices from the specified vertex index to the root.
`+printPath(index: int): void`	Displays a path from the root to the specified vertex.
`+printTree(): void`	Displays tree with the root and all edges.

FIGURE 28.11 The `SearchTree` class describes the nodes with parent–child relationships.

28.7 Depth-First Search (DFS)

Key Point

The depth-first search of a graph starts from a vertex in the graph and visits all vertices in the graph as far as possible before backtracking.

The depth-first search of a graph is like the depth-first search of a tree discussed in Section 25.2.4, Tree Traversal. In the case of a tree, the search starts from the root. In a graph, the search can start from any vertex.

A depth-first search of a tree first visits the root, then recursively visits the subtrees of the root. Similarly, the depth-first search of a graph first visits a vertex, then it recursively visits all the vertices adjacent to that vertex. The difference is that the graph may contain cycles, which could lead to an infinite recursion. To avoid this problem, you need to track the vertices that have already been visited.

The search is called *depth-first* because it searches "deeper" in the graph as much as possible. The search starts from some vertex *v*. After visiting *v*, it visits an unvisited neighbor of *v*. If *v* has no unvisited neighbor, the search backtracks to the vertex from which it reached *v*. We assume that the graph is connected and the search starting from any vertex can reach all the vertices. If this is not the case, see Programming Exercise 28.4 for finding connected components in a graph.

28.7.1 Depth-First Search Algorithm

The algorithm for the depth-first search is described in Listing 28.8.

LISTING 28.8 Depth-First Search Algorithm

```
Input: G = (V, E) and a starting vertex v
Output: a DFS tree rooted at v
```

visit v

check a neighbor

set a parent in the tree

recursive search

```
SearchTree dfs(vertex v) {
  visit v;
  for each neighbor w of v
    if (w has not been visited) {
      set v as the parent for w in the tree;
      dfs(w);
    }
}
```

You can use an array named `isVisited` to denote whether a vertex has been visited. Initially, `isVisited[i]` is `false` for each vertex *i*. Once a vertex, say *v*, is visited, `isVisited[v]` is set to `true`.

Consider the graph in Figure 28.12a. Suppose we start the depth-first search from vertex 0. First visit `0`, then any of its neighbors, say 1. Now `1` is visited, as shown in Figure 28.12b. Vertex 1 has three neighbors—0, 2, and 4. Since 0 has already been visited, you will visit either 2 or 4. Let us pick 2. Now `2` is visited, as shown in Figure 28.12c. Vertex 2 has three neighbors: 0, 1, and 3. Since 0 and 1 have already been visited, pick 3. `3` is now visited, as shown in Figure 28.12d. At this point, the vertices have been visited in this order:

`0, 1, 2, 3`

Since all the neighbors of 3 have been visited, backtrack to 2. Since all the vertices of 2 have been visited, backtrack to 1. 4 is adjacent to 1, but 4 has not been visited. Therefore, visit `4`, as shown in Figure 28.12e. Since all the neighbors of 4 have been visited, backtrack to 1. Since all the neighbors of 1 have been visited, backtrack to 0. Since all the neighbors of 0 have been visited, the search ends.

DFS time complexity

Since each edge and each vertex is visited only once, the time complexity of the `dfs` method is `O(|E| + |V|)`, where `|E|` denotes the number of edges and `|V|` the number of vertices.

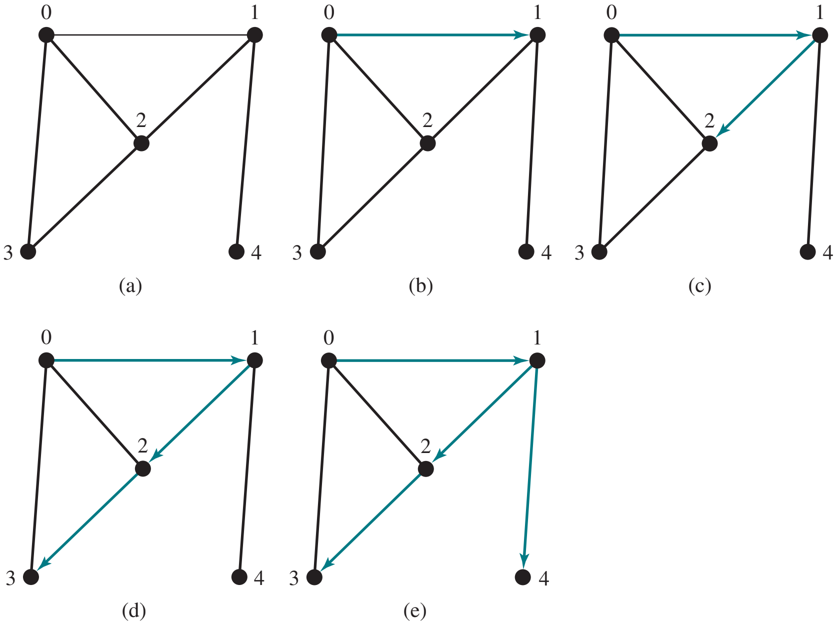


FIGURE 28.12 Depth-first search visits a node and its neighbors recursively.

28.7.2 Implementation of Depth-First Search

The algorithm for DFS in Listing 28.8 uses recursion. It is natural to use recursion to implement it. Alternatively, you can use a stack (see Programming Exercise 28.3).

The `dfs(int v)` method is implemented in lines 148–178 in Listing 28.4. It returns an instance of the `SearchTree` class with vertex `v` as the root. The method stores the vertices searched in the list `searchOrder` (line 150), the parent of each vertex in the array `parent` (line 150), and uses the `isVisited` array to indicate whether a vertex has been visited (line 155). It invokes the helper method `dfs(v, parent, searchOrder, isVisited)` to perform a depth-first search (line 159).

In the recursive helper method, the search starts from vertex `v`. `v` is added to `searchOrder` in line 169 and is marked as visited (line 170). For each unvisited neighbor of `v`, the method is recursively invoked to perform a depth-first search. When a vertex `w` (`w` is `e.v` in line 172 in

Listing 28.4) is visited, the parent of `w` is stored in `parent[w]` (line 174). The method returns when all vertices are visited for a connected graph, or in a connected component.

Listing 28.9 gives a test program that displays a DFS for the graph in Figure 28.1 starting from Chicago. The graphical illustration of the DFS starting from Chicago is shown in Figure 28.13.

U.S. Map Search

LISTING 28.9 `TestDFS.java`

vertices

edges

create a graph

get DFS

get search order

```java
public class TestDFS {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    int[][] edges = {
      {0, 1}, {0, 3}, {0, 5},
      {1, 0}, {1, 2}, {1, 3},
      {2, 1}, {2, 3}, {2, 4}, {2, 10},
      {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
      {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
      {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
      {6, 5}, {6, 7},
      {7, 4}, {7, 5}, {7, 6}, {7, 8},
      {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
      {9, 8}, {9, 11},
      {10, 2}, {10, 4}, {10, 8}, {10, 11},
      {11, 8}, {11, 9}, {11, 10}
    };

    Graph<String> graph = new UnweightedGraph<>(vertices, edges);
    UnweightedGraph<String>.SearchTree dfs =
      graph.dfs(graph.getIndex("Chicago"));

    java.util.List<Integer> searchOrders = dfs.getSearchOrder();
    System.out.println(dfs.getNumberOfVerticesFound() +
      " vertices are searched in this DFS order:");
    for (int i = 0; i < searchOrders.size(); i++)
      System.out.print(graph.getVertex(searchOrders.get(i)) + " ");
    System.out.println();

    for (int i = 0; i < searchOrders.size(); i++)
      if (dfs.getParent(i) != -1)
        System.out.println("parent of " + graph.getVertex(i) +
          " is " + graph.getVertex(dfs.getParent(i)));
  }
}
```

```
12 vertices are searched in this DFS order:
  Chicago Seattle San Francisco Los Angeles Denver
  Kansas City New York Boston Atlanta Miami Houston Dallas
parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is San Francisco
parent of Denver is Los Angeles
parent of Kansas City is Denver
parent of Boston is New York
parent of New York is Kansas City
parent of Atlanta is New York
parent of Miami is Atlanta
parent of Dallas is Houston
parent of Houston is Miami
```

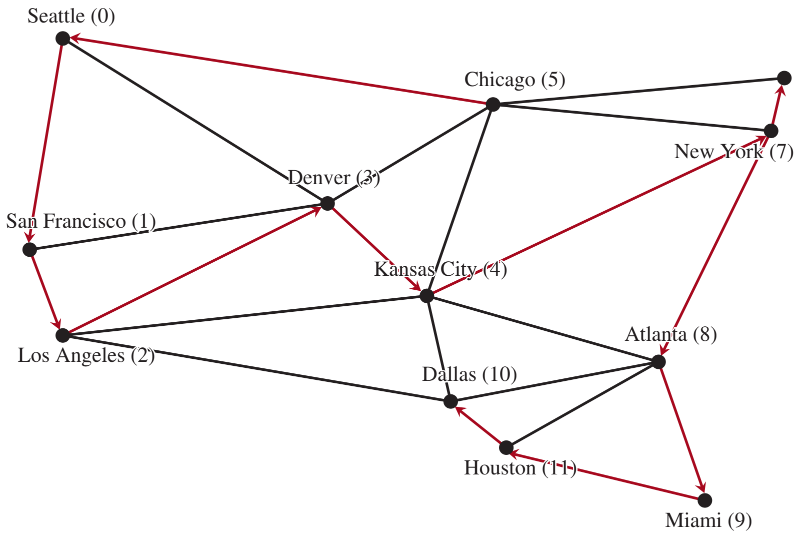


FIGURE 28.13 A DFS search starts from Chicago. *Source*: © Mozilla Firefox.

28.7.3 Applications of the DFS

The depth-first search can be used to solve many problems, such as the following:

- Detecting whether a graph is connected. Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected. Otherwise, the graph is not connected. (See Programming Exercise 28.1.)
- Detecting whether there is a path between two vertices (see Programming Exercise 28.5).
- Finding a path between two vertices (see Programming Exercise 28.5).
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path (see Programming Exercise 28.4).
- Detecting whether there is a cycle in the graph (see Programming Exercise 28.6).
- Finding a cycle in the graph (see Programming Exercise 28.7).
- Finding a Hamiltonian path/cycle. A *Hamiltonian path* in a graph is a path that visits each vertex in the graph exactly once. A *Hamiltonian cycle* visits each vertex in the graph exactly once and returns to the starting vertex (see Programming Exercise 28.17).

The first six problems can be easily solved using the `dfs` method in Listing 28.4. To find a Hamiltonian path/cycle, you have to explore all possible DFSs to find the one that leads to the longest path. The Hamiltonian path/cycle has many applications, including for solving the well-known Knight's Tour problem, which is presented in Supplement VI.E on the Companion Website.

Check Point

28.7.1 What is depth-first search?

28.7.2 Draw a DFS tree for the graph in Figure 28.3b starting from node `A`.

28.7.3 Draw a DFS tree for the graph in Figure 28.1 starting from vertex `Atlanta`.

28.7.4 What is the return type from invoking `dfs(v)`?

28.7.5 The depth-first search algorithm described in Listing 28.8 uses recursion. Alternatively, you can use a stack to implement it, as shown below. Point out the error in this algorithm and give a correct algorithm.

```
// Wrong version
SearchTree dfs(vertex v) {
  push v into the stack;
  mark v visited;

  while (the stack is not empty) {
    pop a vertex, say u, from the stack
    visit u;
    for each neighbor w of u
      if (w has not been visited)
        push w into the stack;
  }
}
```

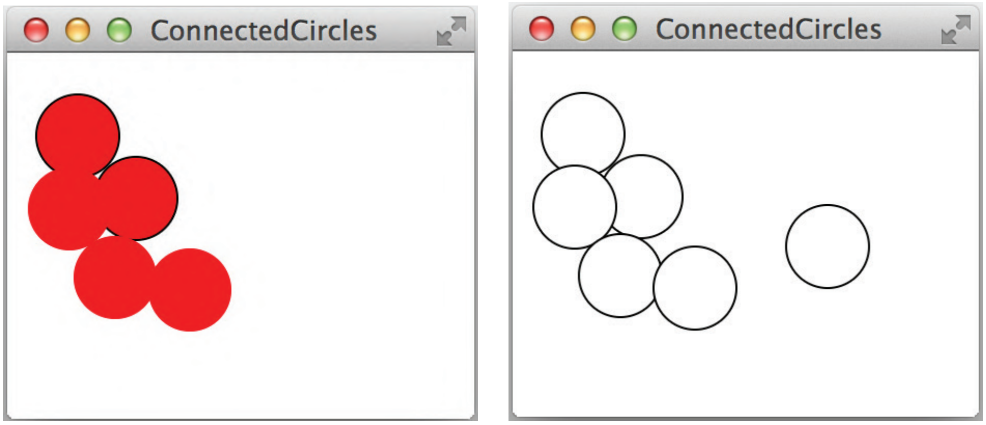
28.8 Case Study: The Connected Circles Problem

Key Point

The connected circles problem is to determine whether all circles in a two-dimensional plane are connected. This problem can be solved using a depth-first traversal.

The DFS algorithm has many applications. This section applies the DFS algorithm to solve the connected circles problem.

In the connected circles problem, you determine whether all the circles in a two-dimensional plane are connected. If all the circles are connected, they are painted as filled circles, as shown in Figure 28.14a. Otherwise, they are not filled, as shown in Figure 28.14b.



(a) Circles are connected (b) Circles are not connected

Figure 28.14 You can apply DFS to determine whether the circles are connected. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

We will write a program that lets the user create a circle by clicking a mouse in a blank area that is not currently covered by a circle. As the circles are added, the circles are repainted filled if they are connected or unfilled otherwise.

We will create a graph to model the problem. Each circle is a vertex in the graph. Two circles are connected if they overlap. We apply the DFS in the graph, and if all vertices are found in the depth-first search, the graph is connected.

The program is given in Listing 28.10.

LISTING 28.10 `ConnectedCircles.java`

```java
import javafx.application.Application;
import javafx.geometry.Point2D;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ConnectedCircles extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place it in the stage
    Scene scene = new Scene(new CirclePane(), 450, 350);
    primaryStage.setTitle("ConnectedCircles"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /** Pane for displaying circles */
  class CirclePane extends Pane {
    public CirclePane() {
      this.setOnMouseClicked(e -> {
        if (!isInsideACircle(new Point2D(e.getX(), e.getY()))) {
          // Add a new circle
          getChildren().add(new Circle(e.getX(), e.getY(), 20));
          colorIfConnected();
        }
      });
    }

    /** Returns true if the point is inside an existing circle */
    private boolean isInsideACircle(Point2D p) {
      for (Node circle: this.getChildren())
        if (circle.contains(p))
          return true;

      return false;
    }

    /** Color all circles if they are connected */
    private void colorIfConnected() {
      if (getChildren().size() == 0)
        return; // No circles in the pane

      // Build the edges
      java.util.List<Edge> edges
        = new java.util.ArrayList<>();
      for (int i = 0; i < getChildren().size(); i++)
        for (int j = i + 1; j < getChildren().size(); j++)
          if (overlaps((Circle)(getChildren().get(i)),
              (Circle)(getChildren().get(j)))) {
            edges.add(new Edge(i, j));
            edges.add(new Edge(j, i));
          }

      // Create a graph with circles as vertices
```

create a circle pane

pane for showing circles

handle mouse clicked

is it inside another circle?

add a new circle

color if all connected

contains the point?

create edges

create a graph
get a search tree
connected?
connected
not connected
two circles overlap?

```
58        Graph<Node> graph = new UnweightedGraph<>(
59          (java.util.List<Node>)getChildren(), edges);
60        UnweightedGraph<Node>.SearchTree tree = graph.dfs(0);
61        boolean isAllCirclesConnected = getChildren().size() == tree
62          .getNumberOfVerticesFound();
63
64        for (Node circle: getChildren()) {
65          if (isAllCirclesConnected) { // All circles are connected
66            ((Circle)circle).setFill(Color.RED);
67          }
68          else {
69            ((Circle)circle).setStroke(Color.BLACK);
70            ((Circle)circle).setFill(Color.WHITE);
71          }
72        }
73      }
74    }
75
76    public static boolean overlaps(Circle circle1, Circle circle2) {
77      return new Point2D(circle1.getCenterX(), circle1.getCenterY()).
78        distance(circle2.getCenterX(), circle2.getCenterY())
79        <= circle1.getRadius() + circle2.getRadius();
80    }
81  }
```

The JavaFX **`Circle`** class contains the data fields **`x`**, **`y`**, and **`radius`**, which specify the circle's center location and radius. It also defines the **`contains`** method for testing whether a point is inside the circle. The **`overlaps`** method (lines 76–80) checks whether two circles overlap.

When the user clicks the mouse outside of any existing circle, a new circle is created centered at the mouse point and the circle is added to the list **`circles`** (line 26).

To detect whether the circles are connected, the program constructs a graph (lines 46–59). The circles are the vertices of the graph. The edges are constructed in lines 47–55. Two circle vertices are connected if they overlap (line 51). The DFS of the graph results in a tree (line 60). The tree's **`getNumberOfVerticesFound()`** returns the number of vertices searched. If it is equal to the number of circles, all circles are connected (lines 61–62).

Check Point

28.8.1 How is a graph created for the connected circles problem?

28.8.2 When you click the mouse inside a circle, does the program create a new circle?

28.8.3 How does the program know if all circles are connected?

28.9 Breadth-First Search (BFS)

Key Point

The breadth-first search of a graph visits the vertices level by level. The first level consists of the starting vertex. Each next level consists of the vertices adjacent to the vertices in the preceding level.

The breadth-first traversal of a graph is like the breadth-first traversal of a tree discussed in Section 25.2.4, Tree Traversal. With breadth-first traversal of a tree, the nodes are visited level by level. First the root is visited, then all the children of the root, then the grandchildren of the root, and so on. Similarly, the breadth-first search of a graph first visits a vertex, then all its adjacent vertices, then all the vertices adjacent to those vertices, and so on. To ensure each vertex is visited only once, it skips a vertex if it has already been visited.

28.9.1 Breadth-First Search Algorithm

The algorithm for the breadth-first search starting from vertex *v* in a graph is described in Listing 28.11.

LISTING 28.11 Breadth-First Search Algorithm

```
Input: G = (V, E) and a starting vertex v
Output: a BFS tree rooted at v

SearchTree bfs(vertex v) {
  create an empty queue for storing vertices to be visited;
  add v into the queue;
  mark v visited;

  while (the queue is not empty) {
    dequeue a vertex, say u, from the queue;
    add u into a list of traversed vertices;
    for each neighbor w of u
      if w has not been visited {
        add w into the queue;
        set u as the parent for w in the tree;
        mark w visited;
      }
  }
}
```

create a queue
enqueue v
dequeue into u
u traversed
check a neighbor w
is w visited?
enqueue w

Consider the graph in Figure 28.15a. Suppose you start the breadth-first search from vertex 0. First visit `0`, then visit all its neighbors, `1`, `2`, and `3`, as shown in Figure 28.15b. Vertex 1 has three neighbors: 0, 2, and 4. Since 0 and 2 have already been visited, you will now visit just `4`, as shown in Figure 28.15c. Vertex 2 has three neighbors, 0, 1, and 3, which have all been visited. Vertex 3 has three neighbors, 0, 2, and 4, which have all been visited. Vertex 4 has two neighbors, 1 and 3, which have all been visited. Hence, the search ends.

Since each edge and each vertex is visited only once, the time complexity of the `bfs` method is `O(|E| + |V|)`, where `|E|` denotes the number of edges and `|V|` the number of vertices.

BFS time complexity

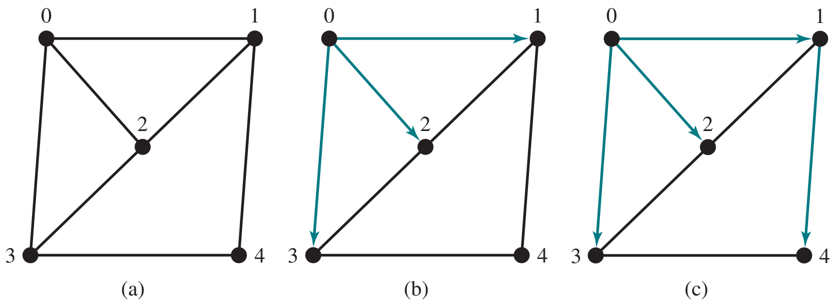


FIGURE 28.15 Breadth-first search visits a node, then its neighbors, then its neighbors's neighbors, and so on.

28.9.2 Implementation of Breadth-First Search

The `bfs(int v)` method is defined in the `Graph` interface and implemented in the `UnweightedGraph` class in Listing 28.4 (lines 182–208). It returns an instance of the `SearchTree` class with vertex `v` as the root. The method stores the vertices searched in the list `searchOrder` (line 183), the parent of each vertex in the array `parent` (line 184), uses a linked list for a queue (lines 188–189), and uses the `isVisited` array to indicate whether a vertex has been visited (line 190). The search starts from vertex `v`. `v` is added to the queue in line 191 and is marked as visited (line 192). The method now examines each vertex `u` in the queue (line 194) and adds it to `searchOrder` (line 196). The method adds each unvisited neighbor `w` (`w` is `e.v` in line 198 in Listing 28.4) of `u` to the queue (line 200), sets its parent to `u` (line 201), and marks it as visited (line 202).

Listing 28.12 gives a test program that displays a BFS for the graph in Figure 28.1 starting from Chicago. The graphical illustration of the BFS starting from Chicago is shown in Figure 28.16.

LISTING 28.12 TestBFS.java

vertices
edges
create a graph
create a BFS tree
get search order

```java
public class TestBFS {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    int[][] edges = {
      {0, 1}, {0, 3}, {0, 5},
      {1, 0}, {1, 2}, {1, 3},
      {2, 1}, {2, 3}, {2, 4}, {2, 10},
      {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
      {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
      {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
      {6, 5}, {6, 7},
      {7, 4}, {7, 5}, {7, 6}, {7, 8},
      {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
      {9, 8}, {9, 11},
      {10, 2}, {10, 4}, {10, 8}, {10, 11},
      {11, 8}, {11, 9}, {11, 10}
    };

    Graph<String> graph = new UnweightedGraph<>(vertices, edges);
    UnweightedGraph<String>.SearchTree bfs =
      graph.bfs(graph.getIndex("Chicago"));

    java.util.List<Integer> searchOrders = bfs.getSearchOrder();
    System.out.println(bfs.getNumberOfVerticesFound() +
      " vertices are searched in this order:");
    for (int i = 0; i < searchOrders.size(); i++)
      System.out.println(graph.getVertex(searchOrders.get(i)));

    for (int i = 0; i < searchOrders.size(); i++)
      if (bfs.getParent(i) != -1)
        System.out.println("parent of " + graph.getVertex(i) +
          " is " + graph.getVertex(bfs.getParent(i)));
  }
}
```

```
12 vertices are searched in this order:
  Chicago Seattle Denver Kansas City Boston New York
  San Francisco Los Angeles Atlanta Dallas Miami Houston
parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is Denver
parent of Denver is Chicago
parent of Kansas City is Chicago
parent of Boston is Chicago
parent of New York is Chicago
parent of Atlanta is Kansas City
parent of Miami is Atlanta
parent of Dallas is Kansas City
parent of Houston is Atlanta
```

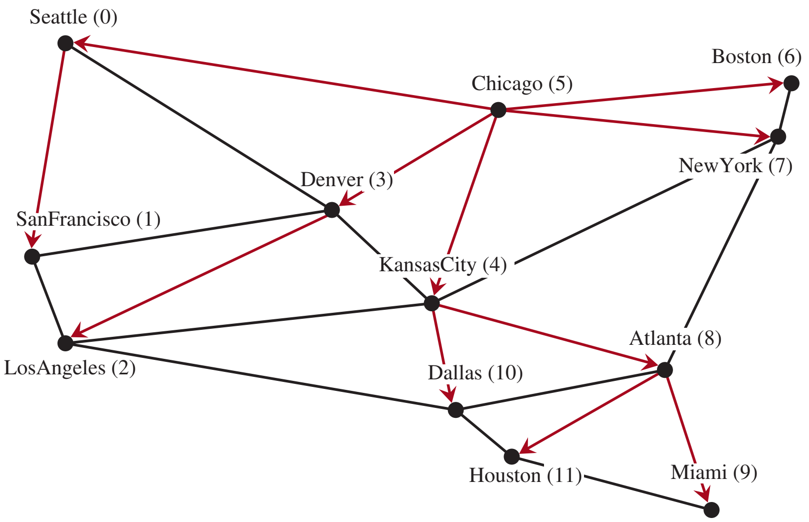


FIGURE 28.16 BFS search starts from Chicago. *Source*: © Mozilla Firefox.

28.9.3 Applications of the BFS

Many of the problems solved by the DFS can also be solved using the BFS. Specifically, the BFS can be used to solve the following problems:

- Detecting whether a graph is connected. A graph is connected if there is a path between any two vertices in the graph.
- Detecting whether there is a path between two vertices.
- Finding the shortest path between two vertices. You can prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node. (See CheckPoint Question 28.9.5.)
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.
- Detecting whether there is a cycle in the graph (see Programming Exercise 28.6).
- Finding a cycle in the graph (see Programming Exercise 28.7).
- Testing whether a graph is bipartite. (A graph is bipartite if the vertices of the graph can be divided into two disjoint sets such that no edges exist between vertices in the same set.) (See Programming Exercise 28.8.)

Check Point

28.9.1 What is the return type from invoking `bfs(v)`?

28.9.2 What is breadth-first search?

28.9.3 Draw a BFS tree for the graph in Figure 28.3b starting from node `A`.

28.9.4 Draw a BFS tree for the graph in Figure 28.1 starting from vertex `Atlanta`.

28.9.5 Prove the path between the root and any node in the BFS tree is the shortest path between the root and the node.

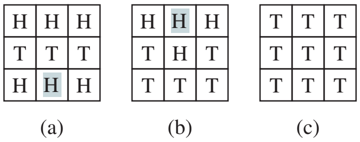
28.10 Case Study: The Nine Tails Problem

Key Point

The nine tails problem can be reduced to the shortest path problem.

The nine tails problem is as follows. Nine coins are placed in a 3 × 3 matrix, with some face up and some face down. A legal move is to take any coin that is face up and reverse it, together with the coins adjacent to it (this does not include coins that are diagonally adjacent). Your task is to find the minimum number of moves that lead to all coins being face down.

For example, start with the nine coins as shown in Figure 28.17a. After you flip the second coin in the last row, the nine coins are now as shown in Figure 28.17b. After you flip the second coin in the first row, the nine coins are all face down, as shown in Figure 28.17c. See liveexample.pearsoncmg.com/dsanimation/NineCoin.html for an interactive demo.



H	H	H
T	T	T
H	H	H

(a)

H	H	H
T	H	T
T	T	T

(b)

T	T	T
T	T	T
T	T	T

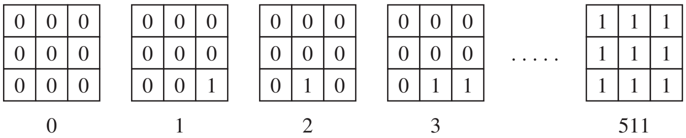
(c)

FIGURE 28.17 The problem is solved when all coins are face down.

We will write a program that prompts the user to enter an initial state of the nine coins and displays the solution, as shown in the following sample run:

```
Enter the initial nine coins Hs and Ts: HHHTTTHHH ↵Enter

The steps to flip the coins are
HHH
TTT
HHH

HHH
THT
TTT

TTT
TTT
TTT
```

Each state of the nine coins represents a node in the graph. For example, the three states in Figure 28.17 correspond to three nodes in the graph. For convenience, we use a 3 × 3 matrix to represent all nodes and use **0** for heads and **1** for tails. Since there are nine cells and each cell is either **0** or **1**, there are a total of 2^9 (512) nodes, labeled **0**, **1**, . . . , and **511**, as shown in Figure 28.18.



0	0	0
0	0	0
0	0	0

0

0	0	0
0	0	0
0	0	1

1

0	0	0
0	0	0
0	1	0

2

0	0	0
0	0	0
0	1	1

3

. . . . .

1	1	1
1	1	1
1	1	1

511

FIGURE 28.18 There are total of 512 nodes labeled in this order: **0**, **1**, **2**, . . . , **511**.

We assign an edge from node **v** to **u** if there is a legal move from **u** to **v**. Figure 28.19 shows a partial graph. Note there is an edge from **511** to **47**, since you can flip a cell in node **47** to become node **511**.

The last node in Figure 28.18 represents the state of nine face-down coins. For convenience, we call this last node the *target node*. Thus, the target node is labeled **511**. Suppose the initial state of the nine tails problem corresponds to the node **s**. The problem is reduced to finding the shortest path from node **s** to the target node, which is equivalent to finding the path from node **s** to the target node in a BFS tree rooted at the target node.

Now the task is to build a directed graph that consists of 512 nodes labeled `0`, `1`, `2`, . . . , `511`, and edges among the nodes. Once the graph is created, obtain a BFS tree rooted at node `511`. From the BFS tree, you can find the shortest path from the root to any vertex. We will create a class named `NineTailModel`, which contains the method to get the shortest path from the target node to any other node. The class UML diagram is shown in Figure 28.19.

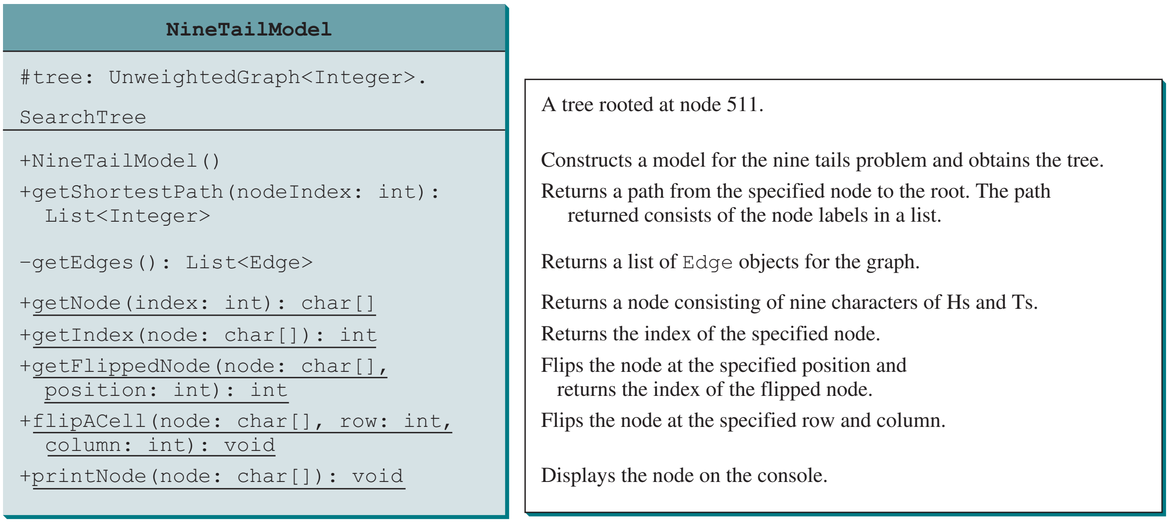

NineTailModel
#tree: UnweightedGraph<Integer>.
SearchTree
+NineTailModel()
+getShortestPath(nodeIndex: int): List<Integer>
-getEdges(): List<Edge>
+getNode(index: int): char[]
+getIndex(node: char[]): int
+getFlippedNode(node: char[], position: int): int
+flipACell(node: char[], row: int, column: int): void
+printNode(node: char[]): void
A tree rooted at node 511.
Constructs a model for the nine tails problem and obtains the tree.
Returns a path from the specified node to the root. The path returned consists of the node labels in a list.
Returns a list of Edge objects for the graph.
Returns a node consisting of nine characters of Hs and Ts.
Returns the index of the specified node.
Flips the node at the specified position and returns the index of the flipped node.
Flips the node at the specified row and column.
Displays the node on the console.

FIGURE 28.19 The `NineTailModel` class models the nine tails problem using a graph.

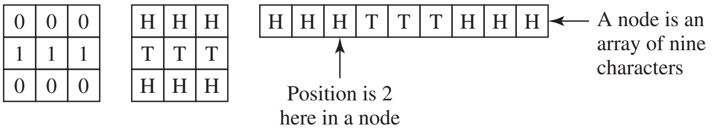
Visually, a node is represented in a 3 × 3 matrix with the letters `H` and `T`. In our program, we use a single-dimensional array of nine characters to represent a node. For example, the node for vertex `1` in Figure 28.18 is represented as {`'H'`, `'H'`, `'H'`, `'H'`, `'H'`, `'H'`, `'H'`, `'H'`, `'T'`} in the array.

The `getEdges()` method returns a list of `Edge` objects.

The `getNode(index)` method returns the node for the specified index. For example, `getNode(0)` returns the node that contains nine `H`s. `getNode(511)` returns the node that contains nine `T`s. The `getIndex(node)` method returns the index of the node.

Note the data field `tree` is defined as protected so it can be accessed from the `WeightedNineTail` subclass in the next chapter.

The `getFlippedNode(char[] node, int position)` method flips the node at the specified position and its adjacent positions. This method returns the index of the new node. The position is a value from 0 to 8, which points to a coin in the node, as shown in the following figure.


0 0 0
1 1 1
0 0 0
H H H
T T T
H H H
H H H T T T H H H
A node is an array of nine characters
Position is 2 here in a node

For example, for node `56` in Figure 28.20, flip it at position `0`, and you will get node `51`. If you flip node `56` at position `1`, you will get node `47`.

The `flipACell(char[] node, int row, int column)` method flips a node at the specified row and column. For example, if you flip node `56` at row `0` and column `0`, the new node is `408`. If you flip node `56` at row `2` and column `0`, the new node is `30`.

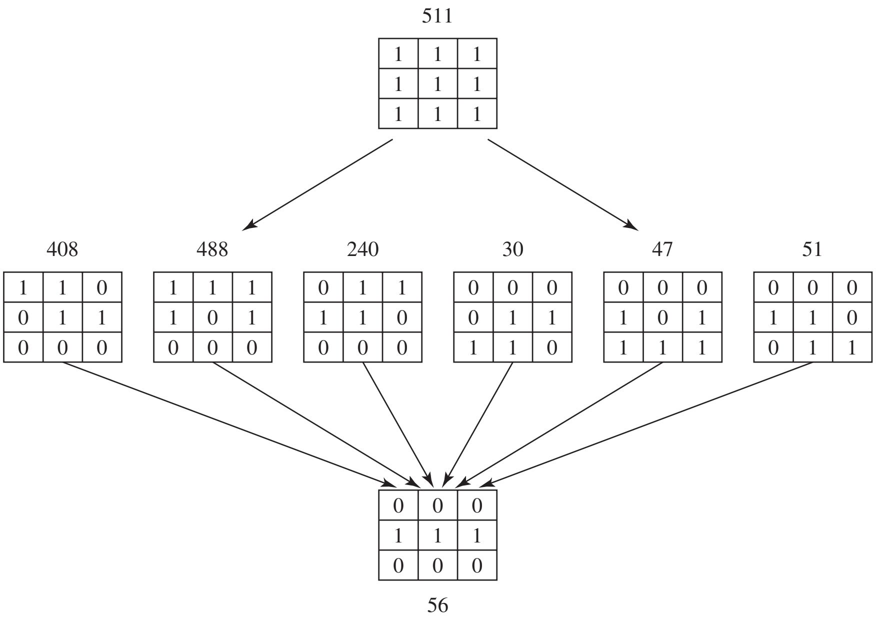

511
1 1 1
1 1 1
1 1 1
408
1 1 0
0 1 1
0 0 0
488
1 1 1
1 0 1
0 0 0
240
0 1 1
1 1 0
0 0 0
30
0 0 0
0 1 1
1 1 0
47
0 0 0
1 0 1
1 1 1
51
0 0 0
1 1 0
0 1 1
0 0 0
1 1 1
0 0 0
56

FIGURE 28.20 If node **u** becomes node **v** after cells are flipped, assign an edge from **v** to **u**.

Listing 28.13 shows the source code for NineTailModel.java.

LISTING 28.13 NineTailModel.java

```java
import java.util.*;

public class NineTailModel {
  public final static int NUMBER_OF_NODES = 512;
  protected UnweightedGraph<Integer>.SearchTree tree;

  /** Construct a model */
  public NineTailModel() {
    // Create edges
    List<Edge> edges = getEdges();

    // Create a graph
    UnweightedGraph<Integer> graph = new UnweightedGraph<>(
      edges, NUMBER_OF_NODES);

    // Obtain a BSF tree rooted at the target node
    tree = graph.bfs(511);
  }

  /** Create all edges for the graph */
  private List<Edge> getEdges() {
    List<Edge> edges =
      new ArrayList<>(); // Store edges

    for (int u = 0; u < NUMBER_OF_NODES; u++) {
      for (int k = 0; k < 9; k++) {
        char[] node = getNode(u); // Get the node for vertex u
        if (node[k] == 'H') {
```

declare a tree

create edges

create graph

create tree

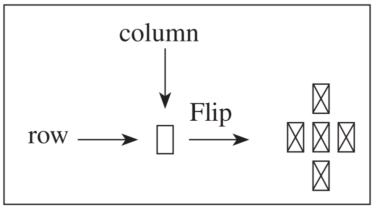
get edges

add an edge

```
          int v = getFlippedNode(node, k);
          // Add edge (v, u) for a legal move from node u to node v
          edges.add(new Edge(v, u));
        }
      }
    }

    return edges;
  }

```

flip cells

```
  public static int getFlippedNode(char[] node, int position) {
    int row = position / 3;
    int column = position % 3;

    flipACell(node, row, column);
    flipACell(node, row - 1, column);
    flipACell(node, row + 1, column);
    flipACell(node, row, column - 1);
    flipACell(node, row, column + 1);

    return getIndex(node);
  }

```



flip a cell

```
  public static void flipACell(char[] node, int row, int column) {
    if (row >= 0 && row <= 2 && column >= 0 && column <= 2) {
      // Within the boundary
      if (node[row * 3 + column] == 'H')
        node[row * 3 + column] = 'T'; // Flip from H to T
      else
        node[row * 3 + column] = 'H'; // Flip from T to H
    }
  }

```

get index for a node

```
  public static int getIndex(char[] node) {
    int result = 0;

    for (int i = 0; i < 9; i++)
      if (node[i] == 'T')
        result = result * 2 + 1;
      else
        result = result * 2 + 0;

    return result;
  }

```

For example:
index: 3
node: HHHHHHHTT

H H H
H H H
H T T

get node for an index

```
  public static char[] getNode(int index) {
    char[] result = new char[9];

    for (int i = 0; i < 9; i++) {
      int digit = index % 2;
      if (digit == 0)
        result[8 - i] = 'H';
      else
        result[8 - i] = 'T';
      index = index / 2;
    }

    return result;
  }

```

For example:
node: THHHHHHTT
index: 259

T H H
H H H
H T T

shortest path

```
  public List<Integer> getShortestPath(int nodeIndex) {
```

display a node

```
 90       return tree.getPath(nodeIndex);
 91     }
 92
 93     public static void printNode(char[] node) {
 94       for (int i = 0; i < 9; i++)
 95         if (i % 3 != 2)
 96           System.out.print(node[i]);
 97         else
 98           System.out.println(node[i]);
 99
100       System.out.println();
101     }
102   }
```

For example:

node: THHHHHHTT

Output:

```
T H H
H H H
H T T
```

The constructor (lines 8–18) creates a graph with 512 nodes, and each edge corresponds to the move from one node to the other (line 10). From the graph, a BFS tree rooted at the target node `511` is obtained (line 17).

To create edges, the `getEdges` method (lines 21–37) checks each node `u` to see if it can be flipped to another node `v`. If so, add (`v`, `u`) to the `Edge` list (line 31). The `getFlippedNode(node, position)` method finds a flipped node by flipping an `H` cell and its neighbors in a node (lines 43–47). The `flipACell(node, row, column)` method actually flips an `H` cell and its neighbors in a node (lines 52–60).

The `getIndex(node)` method is implemented in the same way as converting a binary number to a decimal number (lines 62–72). The `getNode(index)` method returns a node consisting of the letters `H` and `T` (lines 74–87).

The `getShortestpath(nodeIndex)` method invokes the `getPath(nodeIndex)` method to get the vertices in a shortest path from the specified node to the target node (lines 89–91).

The `printNode(node)` method displays a node on the console (lines 93–101).

Listing 28.14 gives a program that prompts the user to enter an initial node and displays the steps to reach the target node.

LISTING 28.14 NineTail.java

initial node

create model

get shortest path

```
 1  import java.util.Scanner;
 2
 3  public class NineTail {
 4    public static void main(String[] args) {
 5      // Prompt the user to enter nine coins' Hs and Ts
 6      System.out.print("Enter the initial nine coins Hs and Ts: ");
 7      Scanner input = new Scanner(System.in);
 8      String s = input.nextLine();
 9      char[] initialNode = s.toCharArray();
10
11      NineTailModel model = new NineTailModel();
12      java.util.List<Integer> path =
13        model.getShortestPath(NineTailModel.getIndex(initialNode));
14
15      System.out.println("The steps to flip the coins are ");
16      for (int i = 0; i < path.size(); i++)
17        NineTailModel.printNode(
18          NineTailModel.getNode(path.get(i).intValue()));
19    }
20  }
```

The program prompts the user to enter an initial node with nine letters with a combination of `H`s and `T`s as a string in line 8, obtains an array of characters from the string (line 9), creates a graph model to get a BFS tree (line 11), obtains the shortest path from the initial node to the target node (lines 12–13), and displays the nodes in the path (lines 16–18).

28.10.1 How are the nodes created for the graph in `NineTailModel`?

28.10.2 How are the edges created for the graph in `NineTailModel`?

28.10.3 What is returned after invoking `getIndex("HTHTTTHHH".toCharArray())` in Listing 28.13? What is returned after invoking `getNode(46)` in Listing 28.13?

28.10.4 If lines 26 and 27 are swapped in Listing 28.13, NineTailModel.java, will the program work? Why not?

KEY TERMS

adjacency list	1053	parallel edge	1048
adjacency matrix	1052	Seven Bridges of Königsberg	1046
adjacent vertices	1048	simple graph	1048
breadth-first search	1067	spanning tree	1048
complete graph	1048	strongly connected graph	1048
cycle	1048	tree	1048
degree	1048	undirected graph	1047
depth-first search	1067	unweighted graph	1048
directed graph	1047	weakly connected graph	1048
graph	1046	weighted graph	1048
incident edges	1048		

CHAPTER SUMMARY

1. A *graph* is a useful mathematical structure that represents relationships among entities in the real world. You learned how to model graphs using classes and interfaces, how to represent vertices and edges using arrays and linked lists, and how to implement operations for graphs.
2. Graph traversal is the process of visiting each vertex in the graph exactly once. You learned two popular ways for traversing a graph: the *depth-first search* (DFS) and *breadth-first search* (BFS).
3. DFS and BFS can be used to solve many problems such as detecting whether a graph is connected, detecting whether there is a cycle in the graph, and finding the shortest path between two vertices.

QUIZ

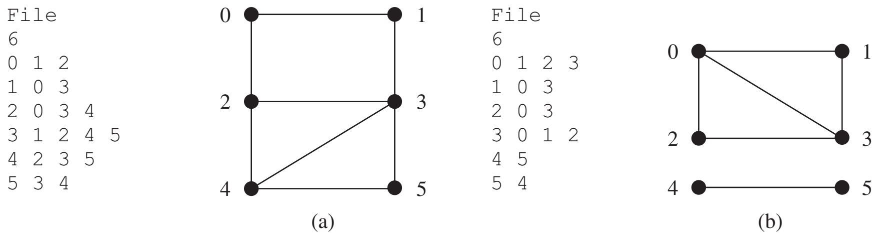
Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

Sections 28.6–28.10

***28.1** (*Test whether a graph is connected*) Write a program that reads a graph from a file and determines whether the graph is connected. The first line in the file contains a number that indicates the number of vertices (`n`). The vertices are labeled as `0`, `1`, . . . , `n−1`. Each subsequent line, with the format `u v1 v2 ...`, describes edges (`u`, `v1`), (`u`, `v2`), and so on. Figure 28.21 gives the examples of two files for their corresponding graphs.



```
File
6
0 1 2
1 0 3
2 0 3 4
3 1 2 4 5
4 2 3 5
5 3 4
```

```
File
6
0 1 2 3
1 0 3
2 0 3
3 0 1 2
4 5
5 4
```

FIGURE 28.21 The vertices and edges of a graph can be stored in a file.

Your program should prompt the user to enter a URL for the file, then it should read data from the file, create an instance `g` of `UnweightedGraph`, invoke `g.printEdges()` to display all edges, and invoke `dfs()` to obtain an instance `tree` of `UnweightedGraph<V>.SearchTree`. If `tree.getNumberOfVerticesFound()` is the same as the number of vertices in the graph, the graph is connected. Here is a sample run of the program:

```
Enter a URL:
https://liveexample.pearsoncmg.com/test/GraphSample1.txt ↵Enter
The number of vertices is 6
Vertex 0: (0, 1) (0, 2)
Vertex 1: (1, 0) (1, 3)
Vertex 2: (2, 0) (2, 3) (2, 4)
Vertex 3: (3, 1) (3, 2) (3, 4) (3, 5)
Vertex 4: (4, 2) (4, 3) (4, 5)
Vertex 5: (5, 3) (5, 4)
The graph is connected
```

(*Hint*: Use `new UnweightedGraph(list, numberOfVertices)` to create a graph, where `list` contains a list of `Edge` objects. Use `new Edge(u, v)` to create an edge. Read the first line to get the number of vertices. Read each subsequent line into a string `s` and use `s.split("[\\;\\s+]")` to extract the vertices from the string and create edges from the vertices.)

***28.2** (*Create a file for a graph*) Modify Listing 28.2, TestGraph.java, to create a file representing `graph1`. The file format is described in Programming Exercise 28.1. Create the file from the array defined in lines 8–21 in Listing 28.2. The number of vertices for the graph is `12`, which will be stored in the first line of the file. The contents of the file should be as follows:

```
12
0 1 3 5
1 0 2 3
2 1 3 4 10
3 0 1 2 4 5
4 2 3 5 7 8 10
5 0 3 4 6 7
6 5 7
7 4 5 6 8
8 4 7 9 10 11
9 8 11
10 2 4 8 11
11 8 9 10
```

***28.3** (*Implement DFS using a stack*) The depth-first search algorithm described in Listing 28.8, Depth-First Search Algorithm, uses recursion. Design a new algorithm without using recursion. Describe it using pseudocode. Implement it by defining a new class named `UnweightedGraphWithNonrecursiveDFS` that extends `UnweightedGraph` and overriding the `dfs` method. Write a test program same as Listing 28.9, TestPFS.java, except that `UnweightedGraph` is replaced by `UnweightedGraphWithNonrecursiveDFS`.

***28.4** (*Find connected components*) Create a new class named `MyGraph` as a subclass of `UnweightedGraph` that contains a method for finding all connected components in a graph with the following header:

```
public List<List<Integer>> getConnectedComponents();
```

The method returns a `List<List<Integer>>`. Each element in the list is another list that contains all the vertices in a connected component. For example, for the graph in Figure 28.21b, `getConnectedComponents()` returns `[[0, 1, 2, 3], [4, 5]]`.

***28.5** (*Find paths*) Define a new class named `UnweightedGraphWithGetPath` that extends `UnweightedGraph` with a new method for finding a path between two vertices with the following header:

```
public List<Integer> getPath(int u, int v);
```

The method returns a `List<Integer>` that contains all the vertices in a path from `u` to `v` in this order. Using the BFS approach, you can obtain the shortest path from `u` to `v`. If there isn't a path from `u` to `v`, the method returns `null`. Write a test program that creates a graph for Figure 28.1. The program prompts the user to enter two cities and displays their paths. Use https://liveexample.pearsoncmg.com/test/Exercise28_05.txt to test your code. Here is a sample run:

```
Enter a starting city: Seattle ↵Enter
Enter an ending city: Miami ↵Enter
The path is Seattle Denver Kansas City Atlanta Miami
```

***28.6** (*Detect cycles*) Define a new class named `UnweightedGraphDetectCycle` that extends `UnweightedGraph` with a new method for determining whether there is a cycle in the graph with the following header:

```
public boolean isCyclic();
```

Describe the algorithm in pseudocode and implement it. Note the graph may be a directed graph.

***28.7** (*Find a cycle*) Define a new class named `UnweightedGraphFindCycle` that extends `UnweightedGraph` with a new method for finding a cycle starting at vertex `u` with the following header:

```
public List<Integer> getACycle(int u);
```

The method returns a `List` that contains all the vertices in a cycle starting from `u`. If the graph doesn't have any cycles, the method returns `null`. Describe the algorithm in pseudocode and implement it.

****28.8** (*Test bipartite*) Recall that a graph is bipartite if its vertices can be divided into two disjoint sets such that no edges exist between vertices in the same set. Define a new class named `UnweightedGraphTestBipartite` with the following method to detect whether the graph is bipartite:

```
public boolean isBipartite();
```

****28.9** (*Get bipartite sets*) Add a new method in `UnweightedGraph` with the following header to return two bipartite sets if the graph is bipartite:

```
public List<List<Integer>> getBipartite();
```

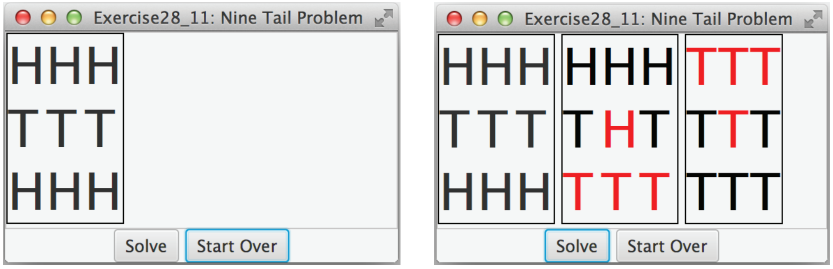
The method returns a `List` that contains two sublists, each of which contains a set of vertices. If the graph is not bipartite, the method returns `null`.

***28.10** (*Find the shortest path*) Write a program that reads a connected graph from a file. The graph is stored in a file using the same format specified in Programming Exercise 28.1. Your program should prompt the user to enter the name of the file, then two vertices, and should display the shortest path between the two vertices. For example, for the graph in Figure 28.21a, the shortest path between `0` and `5` may be displayed as `0 1 3 5`.

Here is a sample run of the program:

```
Enter a file name: c:\exercise\GraphSample1.txt
Enter two vertices (integer indexes): 0 5
The number of vertices is 6
Vertex 0: (0, 1) (0, 2)
Vertex 1: (1, 0) (1, 3)
Vertex 2: (2, 0) (2, 3) (2, 4)
Vertex 3: (3, 1) (3, 2) (3, 4) (3, 5)
Vertex 4: (4, 2) (4, 3) (4, 5)
Vertex 5: (5, 3) (5, 4)
The path is 0 1 3 5
```

****28.11** (*Revise Listing 28.14, NineTail.java*) The program in Listing 28.14 lets the user enter an input for the nine tails problem from the console and displays the result on the console. Write a program that lets the user set an initial state of the nine coins (see Figure 28.22a) and click the *Solve* button to display the solution, as shown in Figure 28.22b. Initially, the user can click the mouse button to flip a coin. Set a red color on the flipped cells.


Exercise28_11: Nine Tail Problem
Solve
Start Over
Exercise28_11: Nine Tail Problem
Solve
Start Over

HHH
TTT
HHH

HHH HHH TTT
TTT THT TTT
HHH TTT TTT

(a) (b)

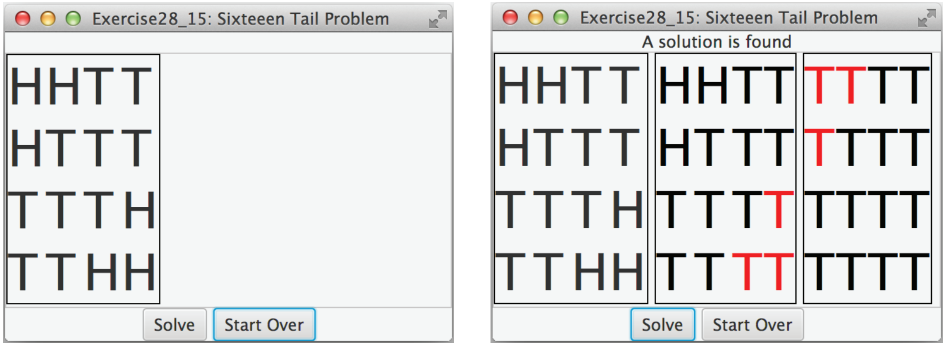
FIGURE 28.22 The program solves the nine tails problem. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****28.12** (*Variation of the nine tails problem*) In the nine tails problem, when you flip a coin, the horizontal and vertical neighboring cells are also flipped. Rewrite the program, assuming all neighboring cells including the diagonal neighbors are also flipped.

****28.13** (*4 × 4 16 tails problem*) Listing 28.14, NineTail.java, presents a solution for the nine tails problem. Revise this program for the 4 × 4 16 tails problem. Note it is possible that a solution may not exist for a starting pattern. If so, report that no solution exists.

****28.14** (*4 × 4 16 tails analysis*) The nine tails problem in the text uses a 3 × 3 matrix. Assume you have 16 coins placed in a 4 × 4 matrix. Write a program to find out the number of the starting patterns that don't have a solution.

***28.15** (*4 × 4 16 tails GUI*) Rewrite Programming Exercise 28.14 to enable the user to set an initial pattern of the 4 × 4 16 tails problem (see Figure 28.23a). The user can click the *Solve* button to display the solution, as shown in Figure 28.23b. Initially, the user can click the mouse button to flip a coin. If a solution does not exist, display a message dialog to report it.


Exercise28_15: Sixteeen Tail Problem
Solve
Start Over
Exercise28_15: Sixteeen Tail Problem
Solve
Start Over

```
HHTT
HTTT
TTTH
TTHH
```

A solution is found

```
HHTT  HHTT  TTTT
HTTT  HTTT  TTTT
TTTH  TTTT  TTTT
TTHH  TTTT  TTTT
```

(a) (b)

FIGURE 28.23 The problem solves the 16 tails problem. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****28.16** (*Induced subgraph*) Given an undirected graph G = (V, E) and an integer *k*, find an induced subgraph H of G of maximum size such that all vertices of H have a degree ≥ *k*, or conclude that no such induced subgraph exists. Implement the method with the following header:

```
public static <V> Graph<V> maxInducedSubgraph(Graph<V> g, int k)
```

The method returns an empty graph if such a subgraph does not exist.

(*Hint*: An intuitive approach is to remove vertices whose degree is less than *k*. As vertices are removed with their adjacent edges, the degrees of other vertices may be reduced. Continue the process until no vertices can be removed, or all the vertices are removed.)

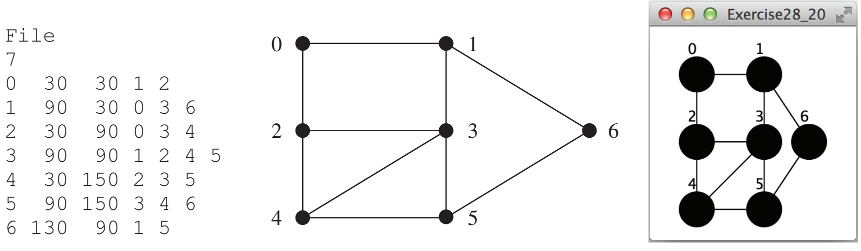
*****28.17** (*Hamiltonian cycle*) The Hamiltonian path algorithm is implemented in Supplement VI.E. Add the following `getHamiltonianCycle` method in the `Graph` interface and implement it in the `UnweightedGraph` class:

```
/** Return a Hamiltonian cycle
  * Return null if the graph doesn't contain a Hamiltonian cycle */
public List<Integer> getHamiltonianCycle()
```

*****28.18** (*Knight's Tour cycle*) Rewrite KnightTourApp.java in the case study in Supplement VI.E to find a knight's tour that visits each square in a chessboard and returns to the starting square. Reduce the Knight's Tour cycle problem to the problem of finding a Hamiltonian cycle.

****28.19** (*Display a DFS/BFS tree in a graph*) Modify `GraphView` in Listing 28.6 to add a new data field `tree` with a setter method. The edges in the tree are displayed in red. Write a program that displays the graph in Figure 28.1 and the DFS/BFS tree starting from a specified city, as shown in Figures 28.13 and 28.16. If a city not in the map is entered, the program displays an error message in the label.

***28.20** (*Display a graph*) Write a program that reads a graph from a file and displays it. The first line in the file contains a number that indicates the number of vertices (`n`). The vertices are labeled `0`, `1`, . . . , `n-1`. Each subsequent line, with the format `u x y v1 v2 ...`, describes the position of `u` at (`x`, `y`) and edges (`u`, `v1`), (`u`, `v2`), and so on. Figure 28.24a gives an example of the file for their corresponding graph. Your program prompts the user to enter the name of the file, reads data from the file, and displays the graph on a pane using `GraphView`, as shown in Figure 28.24b.

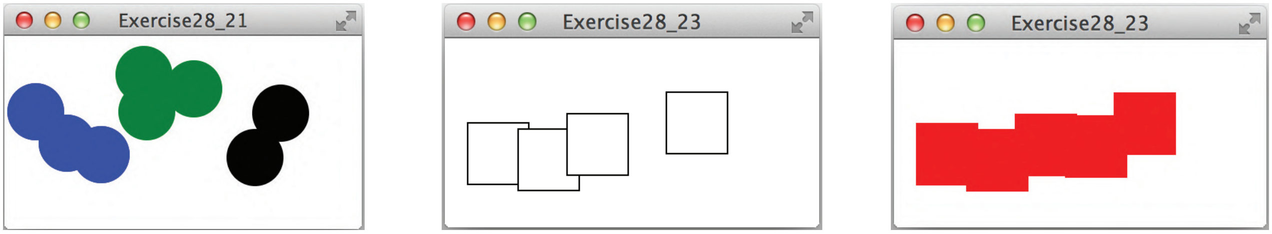


```
File
7
0  30  30 1 2
1  90  30 0 3 6
2  30  90 0 3 4
3  90  90 1 2 4 5
4  30 150 2 3 5
5  90 150 3 4 6
6 130  90 1 5
```

(a) (b)

FIGURE 28.24 The program reads the information about the graph and displays it visually. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****28.21** (*Display sets of connected circles*) Modify Listing 28.10, ConnectedCircles.java to display sets of connected circles in different colors. That is, if two circles are connected, they are displayed using the same color; otherwise, they are not in same color, as shown in Figure 28.25. (*Hint*: See Programming Exercise 28.4.)



(a) (b) (c)

FIGURE 28.25 (a) Connected circles are displayed in the same color. (b) Rectangles are not filled with a color if they are not connected. (c) Rectangles are filled with a color if they are connected. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***28.22** (*Move a circle*) Modify Listing 28.10, ConnectedCircles.java, to enable the user to drag and move a circle.

****28.23** (*Connected rectangles*) Listing 28.10, ConnectedCircles.java, allows the user to create circles and determine whether they are connected. Rewrite the program for rectangles. The program lets the user create a rectangle by clicking a mouse in a blank area that is not currently covered by a rectangle. As the rectangles are added, the rectangles are repainted as filled if they are connected or are unfilled otherwise, as shown in Figure 28.25b–c.

***28.24** (*Remove a circle*) Modify Listing 28.10, ConnectedCircles.java, to enable the user to remove a circle when the mouse is clicked inside the circle.

***28.25** (*Implement remove(V v)*) Modify Listing 28.4, UnweightedGraph.java, to override the `remove(V v)` method defined in the `Graph` interface.

***28.26** (*Implement remove(int u, int v)*) Modify Listing 28.4, UnweightedGraph.java, to override the `remove(int u, int v)` method defined in the `Graph` interface.

CHAPTER 29

WEIGHTED GRAPHS AND APPLICATIONS

Objectives

- To represent weighted edges using adjacency matrices and adjacency lists (§29.2).
- To model weighted graphs using the `WeightedGraph` class that extends the `UnweightedGraph` class (§29.3).
- To design and implement the algorithm for finding a minimum spanning tree (§29.4).
- To define the `MST` class that extends the `SearchTree` class (§29.4).
- To design and implement the algorithm for finding single-source shortest paths (§29.5).
- To define the `ShortestPathTree` class that extends the `SearchTree` class (§29.5).
- To solve the weighted nine tails problem using the shortest-path algorithm (§29.6).

29.1 Introduction

Key Point

A graph is a weighted graph if each edge is assigned a weight. Weighted graphs have many practical applications.

Figure 28.1 assumes the graph represents the number of flights among cities. You can apply the Breadth-First Search (BFS) to find the fewest number of flights between two cities. Assume the edges represent the driving distances among the cities as shown in Figure 29.1. How do you find the minimal total distances for connecting all cities? How do you find the shortest path between two cities? This chapter will address these questions. The former is known as the *minimum spanning tree (MST) problem*, and the latter as the *shortest path problem*.

problem

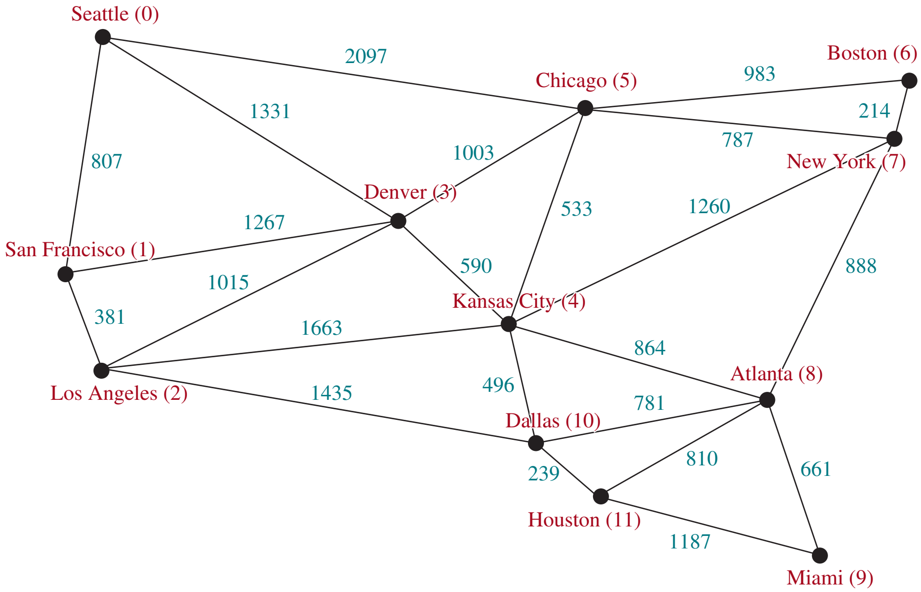

Seattle (0)
2097
Boston (6)
Chicago (5)
983
1331
214
787
807
1003
New York (7)
Denver (3)
533
1260
1267
San Francisco (1)
590
888
1015
Kansas City (4)
381
1663
864
Atlanta (8)
Los Angeles (2)
1435
496
781
Dallas (10)
810
661
239
Houston (11)
1187
Miami (9)

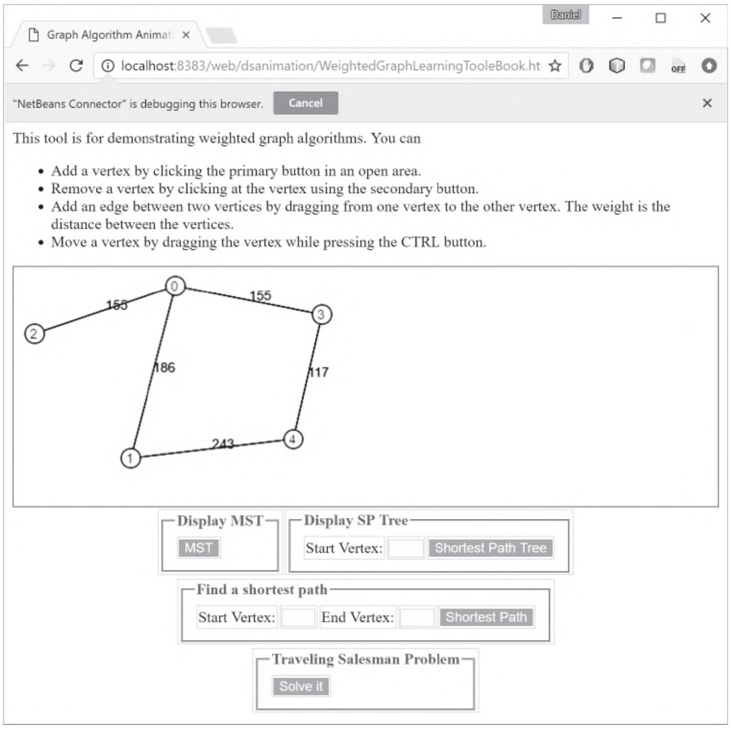
FIGURE 29.1 The graph models the distances among the cities.

The preceding chapter introduced the concept of graphs. You learned how to represent edges using edge arrays, edge lists, adjacency matrices, and adjacency lists, and how to model a graph using the `Graph` interface and the `UnweightedGraph` class. The preceding chapter also introduced two important techniques for traversing graphs: depth-first search and breadth-first search, and applied traversal to solve practical problems. This chapter will introduce weighted graphs. You will learn the algorithm for finding a minimum spanning tree in Section 29.4, and the algorithm for finding shortest paths in Section 29.5.

weighted graph learning tool
on Companion Website

Pedagogical Note

Before we introduce the algorithms and applications for weighted graphs, it is helpful to get acquainted with weighted graphs using the GUI interactive tool, at liveexample.pearsoncmg.com/dsanimation/WeightedGraphLearningTooleBook.html, as shown in Figure 29.2. The tool allows you to enter vertices, create weighted edges, view the graph, and find an MST, all shortest paths from a single source other operations.


Graph Algorithm Animat
Daniel
localhost:8383/web/dsanimation/WeightedGraphLearningTooleBook.ht
OFF
Cancel
0
1
2
3
4
155
155
186
117
243
Display MST
MST
Display SP Tree
Start Vertex:
Shortest Path Tree
Find a shortest path
Start Vertex:
End Vertex:
Shortest Path
Traveling Salesman Problem
Solve it

"NetBeans Connector" is debugging this browser.

This tool is for demonstrating weighted graph algorithms. You can

- Add a vertex by clicking the primary button in an open area.
- Remove a vertex by clicking at the vertex using the secondary button.
- Add an edge between two vertices by dragging from one vertex to the other vertex. The weight is the distance between the vertices.
- Move a vertex by dragging the vertex while pressing the CTRL button.

FIGURE 29.2 You can use the tool to create a weighted graph with mouse gestures and show the MST and shortest paths, and perform other operations. *Source*: © Mozilla Firefox.

29.2 Representing Weighted Graphs

Weighted edges can be stored in adjacency lists.

Key Point

There are two types of weighted graphs: vertex weighted and edge weighted. In a *vertex-weighted graph*, each vertex is assigned a weight. In an *edge-weighted graph*, each edge is assigned a weight. Of the two types, edge-weighted graphs have more applications. This chapter considers edge-weighted graphs.

vertex-weighted graph
edge-weighted graph

Weighted graphs can be represented in the same way as unweighted graphs, except that you have to represent the weights on the edges. As with unweighted graphs, the vertices in weighted graphs can be stored in an array. This section introduces three representations for the edges in weighted graphs.

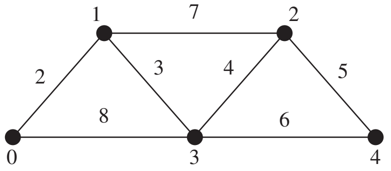
29.2.1 Representing Weighted Edges: Edge Array

Weighted edges can be represented using a two-dimensional array. For example, you can store all the edges in the graph in Figure 29.3a using the array in Figure 29.3b.

Note

Weights can be of any type: `Integer`, `Double`, `BigDecimal`, and so on. You can use a two-dimensional array of the `Object` type to represent weighted edges as follows:

```
Object[][] edges = {
  {Integer.valueOf(0), Integer.valueOf(1), new SomeTypeForWeight(2)},
  {Integer.valueOf(0), Integer.valueOf(3), new SomeTypeForWeight(8)},
  ...
};
```



(a)

vertex weight

```
int[][] edges = {{0, 1, 2}, {0, 3, 8},
  {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
  {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
  {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
  {4, 2, 5}, {4, 3, 6}
};
```

(b)

Figure 29.3 Each edge is assigned a weight in an edge-weighted graph.

29.2.2 Weighted Adjacency Matrices

Assume the graph has *n* vertices. You can use a two-dimensional *n* × *n* matrix, say **`weights`**, to represent the weights on edges. **`weights[i][j]`** represents the weight on edge (**`i`**, **`j`**). If vertices **`i`** and **`j`** are not connected, **`weights[i][j]`** is **`null`**. For example, the weights in the graph in Figure 29.3a can be represented using an adjacency matrix as follows:

```
Integer[][] adjacencyMatrix = {
  {null, 2, null, 8, null},
  {2, null, 7, 3, null},
  {null, 7, null, 4, 5},
  {8, 3, 4, null, 6},
  {null, null, 5, 6, null}
};
```

	0	1	2	3	4
0	null	2	null	8	null
1	2	null	7	3	null
2	null	7	null	4	5
3	8	3	4	null	6
4	null	null	5	6	null

29.2.3 Adjacency Lists

Another way to represent the edges is to define edges as objects. The **`Edge`** class was defined to represent an unweighted edge in Listing 28.3. For weighted edges, we define the **`WeightedEdge`** class as shown in Listing 29.1.

Listing 29.1 WeightedEdge.java

edge weight

constructor

compare edges

```
public class WeightedEdge extends Edge
    implements Comparable<WeightedEdge> {
  public double weight;  // The weight on edge (u, v)

  /** Create a weighted edge on (u, v) */
  public WeightedEdge(int u, int v, double weight) {
    super(u, v);
    this.weight = weight;
  }

  @Override  /** Compare two edges on weights */
  public int compareTo(WeightedEdge edge) {
    if (weight > edge.weight)
      return 1;
    else if (weight == edge.weight)
      return 0;
    else
      return -1;
  }
}
```

An `Edge` object represents an edge from vertex `u` to `v`. `WeightedEdge` extends `Edge` with a new property `weight`. To create a `WeightedEdge` object, use `new WeightedEdge(i, j, w)`, where `w` is the weight on edge (`i`, `j`). Often you need to compare the weights of the edges. For this reason, the `WeightedEdge` class implements the `Comparable` interface.

For unweighted graphs, we use adjacency lists to represent edges. For weighted graphs, we still use adjacency lists, the adjacency lists for the vertices in the graph in Figure 29.3a can be represented as follows:

```
java.util.List<WeightedEdge>[] list = new java.util.List[5];
```

`list[0]`	WeightedEdge(0, 1, 2)	WeightedEdge(0, 3, 8)		
`list[1]`	WeightedEdge(1, 0, 2)	WeightedEdge(1, 3, 3)	WeightedEdge(1, 2, 7)	
`list[2]`	WeightedEdge(2, 3, 4)	WeightedEdge(2, 4, 5)	WeightedEdge(2, 1, 7)	
`list[3]`	WeightedEdge(3, 1, 3)	WeightedEdge(3, 2, 4)	WeightedEdge(3, 4, 6)	WeightedEdge(3, 0, 8)
`list[4]`	WeightedEdge(4, 2, 5)	WeightedEdge(4, 3, 6)		

`list[i]` stores all edges adjacent to vertex `i`.

For flexibility, we will use an array list rather than a fixed-sized array to represent `list` as follows:

```
List<List<WeightedEdge>> list = new java.util.ArrayList<>();
```

Check Point

29.2.1 For the code `WeightedEdge edge = new WeightedEdge(1, 2, 3.5)`, what is `edge.u`, `edge.v`, and `edge.weight`?

29.2.2 What is the output of the following code?

```
List<WeightedEdge> list = new ArrayList<>();
list.add(new WeightedEdge(1, 2, 3.5));
list.add(new WeightedEdge(2, 3, 4.5));
WeightedEdge e = java.util.Collections.max(list);
System.out.println(e.u);
System.out.println(e.v);
System.out.println(e.weight);
```

29.3 The WeightedGraph Class

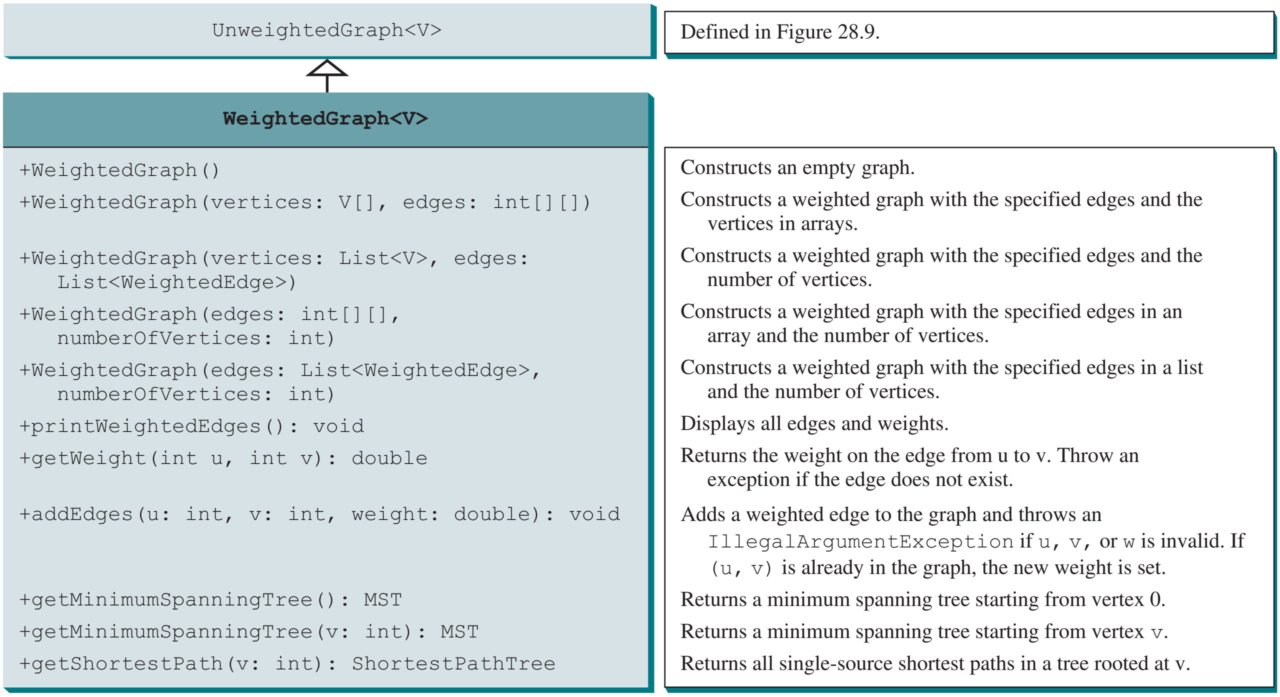
The `WeightedGraph` *class extends* `UnweightedGraph`.

Key Point

The preceding chapter designed the `Graph` interface and the `UnweightedGraph` class for modeling graphs. We now design `WeightedGraph` as a subclass of `UnweightedGraph`, as shown in Figure 29.4.

`WeightedGraph` simply extends `UnweightedGraph` with five constructors for creating concrete `WeightedGraph` instances. `WeightedGraph` inherits all methods from `UnweightedGraph`, overrides the `clear` and `addVertex` methods, implements a new `addEdge` method for adding a weighted edge, and also introduces new methods for obtaining minimum spanning trees and for finding all *single-source shortest paths*. Minimum spanning trees and shortest paths will be introduced in Sections 29.4 and 29.5, respectively.

Listing 29.2 implements `WeightedGraph`. Edge adjacency lists (lines 38–63) are used internally to store adjacent edges for a vertex. When a `WeightedGraph` is constructed, its edge



WeightedGraph<V>	
UnweightedGraph<V>	Defined in Figure 28.9.
+WeightedGraph()	Constructs an empty graph.
+WeightedGraph(vertices: V[], edges: int[][])	Constructs a weighted graph with the specified edges and the vertices in arrays.
+WeightedGraph(vertices: List<V>, edges: List<WeightedEdge>)	Constructs a weighted graph with the specified edges and the number of vertices.
+WeightedGraph(edges: int[][], numberOfVertices: int)	Constructs a weighted graph with the specified edges in an array and the number of vertices.
+WeightedGraph(edges: List<WeightedEdge>, numberOfVertices: int)	Constructs a weighted graph with the specified edges in a list and the number of vertices.
+printWeightedEdges(): void	Displays all edges and weights.
+getWeight(int u, int v): double	Returns the weight on the edge from u to v. Throw an exception if the edge does not exist.
+addEdges(u: int, v: int, weight: double): void	Adds a weighted edge to the graph and throws an `IllegalArgumentException` if `u`, `v`, or `w` is invalid. If `(u, v)` is already in the graph, the new weight is set.
+getMinimumSpanningTree(): MST	Returns a minimum spanning tree starting from vertex 0.
+getMinimumSpanningTree(v: int): MST	Returns a minimum spanning tree starting from vertex `v`.
+getShortestPath(v: int): ShortestPathTree	Returns all single-source shortest paths in a tree rooted at v.

FIGURE 29.4 **`WeightedGraph`** extends **`UnweightedGraph`**.

adjacency lists are created (lines 47 and 57). The methods **`getMinimumSpanningTree()`** (lines 99–138) and **`getShortestPath()`** (lines 156–197) will be introduced in upcoming sections.

LISTING 29.2 WeightedGraph.java

no-arg constructor

constructor

constructor

```
 1 import java.util.*;
 2
 3 public class WeightedGraph<V>extends UnweightedGraph<V> {
 4   /** Construct an empty */
 5   public WeightedGraph() {
 6   }
 7
 8   /** Construct a WeightedGraph from vertices and edged in arrays */
 9   public WeightedGraph(V[] vertices, int[][] edges) {
10     createWeightedGraph(java.util.Arrays.asList(vertices), edges);
11   }
12
13   /** Construct a WeightedGraph from vertices and edges in list */
14   public WeightedGraph(int[][] edges, int numberOfVertices) {
15     List<V> vertices = new ArrayList<>();
16     for (int i = 0; i < numberOfVertices; i++)
17       vertices.add((V)(Integer.valueOf(i)));
18
19     createWeightedGraph(vertices, edges);
20   }
21
```

```
  /** Construct a WeightedGraph for vertices 0, 1, 2 and edge list */
  public WeightedGraph(List<V> vertices, List<WeightedEdge> edges) {
    createWeightedGraph(vertices, edges);
  }

  /** Construct a WeightedGraph from vertices 0, 1, and edge array */
  public WeightedGraph(List<WeightedEdge> edges,
      int numberOfVertices) {
    List<V> vertices = new ArrayList<>();
    for (int i = 0; i < numberOfVertices; i++)
      vertices.add((V)(Integer.valueOf(i)));

    createWeightedGraph(vertices, edges);
  }

  /** Create adjacency lists from edge arrays */
  private void createWeightedGraph(List<V> vertices, int[][] edges) {
    this.vertices = vertices;

    for (int i = 0; i < vertices.size(); i++) {
      neighbors.add(new ArrayList<Edge>()); // Create a list for vertices
    }

    for (int i = 0; i < edges.length; i++) {
      neighbors.get(edges[i][0]).add(
        new WeightedEdge(edges[i][0], edges[i][1], edges[i][2]));
    }
  }

  /** Create adjacency lists from edge lists */
  private void createWeightedGraph(
      List<V> vertices, List<WeightedEdge> edges) {
    this.vertices = vertices;

    for (int i = 0; i < vertices.size(); i++) {
      neighbors.add(new ArrayList<Edge>()); // Create a list for vertices
    }

    for (WeightedEdge edge: edges) {
      neighbors.get(edge.u).add(edge); // Add an edge into the list
    }
  }

  /** Return the weight on the edge (u, v) */
  public double getWeight(int u, int v) throws Exception {
    for (Edge edge : neighbors.get(u)) {
      if (edge.v == v) {
        return ((WeightedEdge)edge).weight;
      }
    }

    throw new Exception("Edge does not exit");
  }

  /** Display edges with weights */
  public void printWeightedEdges() {
    for (int i = 0; i < getSize(); i++) {
      System.out.print(getVertex(i) + " (" + i + "): ");
      for (Edge edge : neighbors.get(i)) {
        System.out.print("(" + edge.u +
```

constructor

constructor

create list for vertices

create a weighted edge

create list for vertices

get edge weight

print edges

add edge
get an MST
start from vertex 0
MST from a starting vertex
initialize cost
initialize parent
minimum spanning tree
expand MST
update total cost
vertex with smallest cost
add to tree
adjust cost
create an MST
MST inner class

```
          ", " + edge.v + ", " + ((WeightedEdge)edge).weight + ") ");
      }
      System.out.println();
    }
  }

  /** Add edges to the weighted graph */
  public boolean addEdge(int u, int v, double weight) {
    return addEdge(new WeightedEdge(u, v, weight));
  }

  /** Get a minimum spanning tree rooted at vertex 0 */
  public MST getMinimumSpanningTree() {
    return getMinimumSpanningTree(0);
  }

  /** Get a minimum spanning tree rooted at a specified vertex */
  public MST getMinimumSpanningTree(int startingVertex) {
    // cost[v] stores the cost by adding v to the tree
    double[] cost = new double[getSize()];
    for (int i = 0; i < cost.length; i++) {
      cost[i] = Double.POSITIVE_INFINITY; // Initial cost
    }
    cost[startingVertex] = 0; // Cost of source is 0

    int[] parent = new int[getSize()]; // Parent of a vertex
    parent[startingVertex] = -1; // startingVertex is the root
    double totalWeight = 0; // Total weight of the tree thus far

    List<Integer> T = new ArrayList<>();

    // Expand T
    while (T.size() < getSize()) {
      // Find smallest cost u in V - T
      int u = -1; // Vertex to be determined
      double currentMinCost = Double.POSITIVE_INFINITY;
      for (int i = 0; i < getSize(); i++) {
        if (!T.contains(i) && cost[i] < currentMinCost) {
          currentMinCost = cost[i];
          u = i;
        }
      }

      if (u == -1) break; else T.add(u); // Add a new vertex to T
      totalWeight += cost[u]; // Add cost[u] to the tree

      // Adjust cost[v] for v that is adjacent to u and v in V - T
      for (Edge e: neighbors.get(u)) {
        if (!T.contains(e.v) && cost[e.v] > ((WeightedEdge)e).weight) {
          cost[e.v] = ((WeightedEdge)e).weight;
          parent[e.v] = u;
        }
      }
    } // End of while

    return new MST(startingVertex, parent, T, totalWeight);
  }

  /** MST is an inner class in WeightedGraph */
  public class MST extends SearchTree {
    private double totalWeight; // Total weight of all edges in the tree

    public MST(int root, int[] parent, List<Integer> searchOrder,
        double totalWeight) {
      super(root, parent, searchOrder);
      this.totalWeight = totalWeight;
    }

    public double getTotalWeight() {
      return totalWeight;
    }
  }

  /** Find single-source shortest paths */
  public ShortestPathTree getShortestPath(int sourceVertex) {
    // cost[v] stores the cost of the path from v to the source
    double[] cost = new double[getSize()];
    for (int i = 0; i < cost.length; i++) {
      cost[i] = Double.POSITIVE_INFINITY; // Initial cost set to infinity
    }
    cost[sourceVertex] = 0; // Cost of source is 0

    // parent[v] stores the previous vertex of v in the path
    int[] parent = new int[getSize()];
    parent[sourceVertex] = -1; // The parent of source is set to -1

    // T stores the vertices whose path found so far
    List<Integer> T = new ArrayList<>();

    // Expand T
    while (T.size() < getSize()) {
      // Find smallest cost u in V - T
      int u = -1; // Vertex to be determined
      double currentMinCost = Double.POSITIVE_INFINITY;
      for (int i = 0; i < getSize(); i++) {
        if (!T.contains(i) && cost[i] < currentMinCost) {
          currentMinCost = cost[i];
          u = i;
        }
      }

      if (u == -1) break; else T.add(u); // Add a new vertex to T

      // Adjust cost[v] for v that is adjacent to u and v in V - T
      for (Edge e: neighbors.get(u)) {
        if (!T.contains(e.v)
            && cost[e.v] > cost[u] + ((WeightedEdge)e).weight) {
          cost[e.v] = cost[u] + ((WeightedEdge)e).weight;
          parent[e.v] = u;
        }
      }
    } // End of while

    // Create a ShortestPathTree
    return new ShortestPathTree(sourceVertex, parent, T, cost);
  }

  /** ShortestPathTree is an inner class in WeightedGraph */
  public class ShortestPathTree extends SearchTree {
    private double[] cost; // cost[v] is the cost from v to source
```

total weight in tree
getShortestPath
initialize cost
shortest-path tree
expand tree
vertex with smallest cost
add to T
adjust cost
adjust parent
create a tree
shortest-path tree
cost

constructor
get cost
print all paths

```
202
203     /** Construct a path */
204     public ShortestPathTree(int source, int[] parent,
205         List<Integer> searchOrder, double[] cost) {
206       super(source, parent, searchOrder);
207       this.cost = cost;
208     }
209
210     /** Return the cost for a path from the root to vertex v */
211     public double getCost(int v) {
212       return cost[v];
213     }
214
215     /** Print paths from all vertices to the source */
216     public void printAllPaths() {
217       System.out.println("All shortest paths from " +
218         vertices.get(getRoot()) + " are:");
219       for (int i = 0; i < cost.length; i++) {
220         printPath(i); // Print a path from i to the source
221         System.out.println("(cost: " + cost[i] + ")"); // Path cost
222       }
223     }
224   }
225 }
```

The `WeightedGraph` class extends the `UnweightedGraph` class (line 3). The properties `vertices` and `neighbors` in `UnweightedGraph` are inherited in `WeightedGraph`. `neighbors` is a list. Each element in the list is another list that contains edges. For unweighted graph, each edge is an instance of `Edge`. For a weighted graph, each edge is an instance of `WeightedEdge`. `WeightedEdge` is a subtype of `Edge`. So you can add a weighted edge into `neighbors.get(i)` for a weighted graph (line 47).

The `addEdge(u, v, weight)` method (lines 88–91) adds an edge (`u`, `v`, `weight`) to the graph. If a graph is undirected, you should invoke `addEdge(u, v, weight)` and `addEdge(v, u, weight)` to add an edge between `u` and `v`.

Listing 29.3 gives a test program that creates a graph for the one in Figure 29.1 and another graph for the one in Figure 29.3a.

LISTING 29.3 TestWeightedGraph.java

vertices
edges

```
public class TestWeightedGraph {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    int[][] edges = {
      {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
      {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
      {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
      {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
        {3, 5, 1003},
      {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
        {4, 8, 864}, {4, 10, 496},
      {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
        {5, 6, 983}, {5, 7, 787},
      {6, 5, 983}, {6, 7, 214},
      {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
      {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
        {8, 10, 781}, {8, 11, 810},
      {9, 8, 661}, {9, 11, 1187},
      {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
      {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
    };

    WeightedGraph<String> graph1 =
      new WeightedGraph<>(vertices, edges);
    System.out.println("The number of vertices in graph1: "
      + graph1.getSize());
    System.out.println("The vertex with index 1 is "
      + graph1.getVertex(1));
    System.out.println("The index for Miami is " +
      graph1.getIndex("Miami"));
    System.out.println("The edges for graph1:");
    graph1.printWeightedEdges();

    edges = new int[][] {
      {0, 1, 2}, {0, 3, 8},
      {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
      {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
      {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
      {4, 2, 5}, {4, 3, 6}
    };
    WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);
    System.out.println("\nThe edges for graph2:");
    graph2.printWeightedEdges();
  }
}
```

create graph

print edges

edges

create graph

print edges

```
The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Vertex 0: (0, 1, 807) (0, 3, 1331) (0, 5, 2097)
Vertex 1: (1, 2, 381) (1, 0, 807) (1, 3, 1267)
Vertex 2: (2, 1, 381) (2, 3, 1015) (2, 4, 1663) (2, 10, 1435)
Vertex 3: (3, 4, 599) (3, 5, 1003) (3, 1, 1267)
  (3, 0, 1331) (3, 2, 1015)
Vertex 4: (4, 10, 496) (4, 8, 864) (4, 5, 533) (4, 2, 1663)
  (4, 7, 1260) (4, 3, 599)
Vertex 5: (5, 4, 533) (5, 7, 787) (5, 3, 1003)
  (5, 0, 2097) (5, 6, 983)
Vertex 6: (6, 7, 214) (6, 5, 983)
Vertex 7: (7, 6, 214) (7, 8, 888) (7, 5, 787) (7, 4, 1260)
Vertex 8: (8, 9, 661) (8, 10, 781) (8, 4, 864)
  (8, 7, 888) (8, 11, 810)
Vertex 9: (9, 8, 661) (9, 11, 1187)
Vertex 10: (10, 11, 239) (10, 4, 496) (10, 8, 781) (10, 2, 1435)
Vertex 11: (11, 10, 239) (11, 9, 1187) (11, 8, 810)

The edges for graph2:
Vertex 0: (0, 1, 2) (0, 3, 8)
Vertex 1: (1, 0, 2) (1, 2, 7) (1, 3, 3)
Vertex 2: (2, 3, 4) (2, 1, 7) (2, 4, 5)
Vertex 3: (3, 1, 3) (3, 4, 6) (3, 2, 4) (3, 0, 8)
Vertex 4: (4, 2, 5) (4, 3, 6)
```

The program creates `graph1` for the graph in Figure 29.1 in lines 3–27. The vertices for `graph1` are defined in lines 3–5. The edges for `graph1` are defined in lines 7–24. The edges are represented using a two-dimensional array. For each row `i` in the array, `edges[i][0]` and `edges[i][1]` indicate there is an edge from vertex `edges[i][0]` to vertex `edges[i][1]` and the weight for the edge is `edges[i][2]`. For example, {`0`, `1`, `807`} (line 8) represents the edge from vertex `0` (`edges[0][0]`) to vertex `1` (`edges[0][1]`) with weight `807` (`edges[0][2]`). {`0`, `5`, `2097`} (line 8) represents the edge from vertex `0` (`edges[2][0]`) to vertex `5` (`edges[2][1]`) with weight `2097` (`edges[2][2]`). Line 35 invokes the `printWeightedEdges()` method on `graph1` to display all edges in `graph1`.

The program creates the edges for `graph2` for the graph in Figure 29.3a in lines 37–44. Line 46 invokes the `printWeightedEdges()` method on `graph2` to display all edges in `graph2`.

Check Point

29.3.1 If a priority queue is used to store weighted edges, what is the output of the following code?

```
PriorityQueue<WeightedEdge> q = new PriorityQueue<>();
q.offer(new WeightedEdge(1, 2, 3.5));
q.offer(new WeightedEdge(1, 6, 6.5));
q.offer(new WeightedEdge(1, 7, 1.5));
System.out.println(q.poll().weight);
System.out.println(q.poll().weight);
System.out.println(q.poll().weight);
```

29.3.2 If a priority queue is used to store weighted edges, what is wrong in the following code? Fix it and show the output.

```
List<PriorityQueue<WeightedEdge>> queues = new ArrayList<>();
queues.get(0).offer(new WeightedEdge(0, 2, 3.5));
queues.get(0).offer(new WeightedEdge(0, 6, 6.5));
queues.get(0).offer(new WeightedEdge(0, 7, 1.5));
queues.get(1).offer(new WeightedEdge(1, 0, 3.5));
queues.get(1).offer(new WeightedEdge(1, 5, 8.5));
queues.get(1).offer(new WeightedEdge(1, 8, 19.5));
System.out.println(queues.get(0).peek()
  .compareTo(queues.get(1).peek()));
```

29.3.3 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    WeightedGraph<Character> graph = new WeightedGraph<>();
    graph.addVertex('U');
    graph.addVertex('V');
    int indexForU = graph.getIndex('U');
    int indexForV = graph.getIndex('V');
    System.out.println("indexForU is " + indexForU);
    System.out.println("indexForV is " + indexForV);
    graph.addEdge(indexForU, indexForV, 2.5);
    System.out.println("Degree of U is " +
      graph.getDegree(indexForU));
    System.out.println("Degree of V is " +
      graph.getDegree(indexForV));
    System.out.println("Weight of UV is " +
      graph.getWeight(indexForU, indexOfV));
  }
}
```

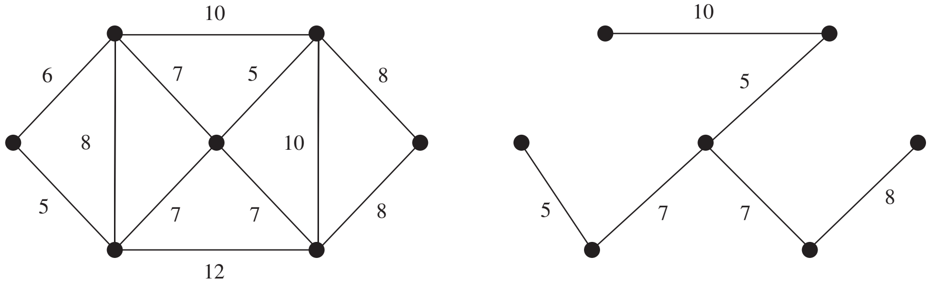
29.4 Minimum Spanning Trees

Key Point

A minimum spanning tree of a graph is a spanning tree with the minimum total weights.

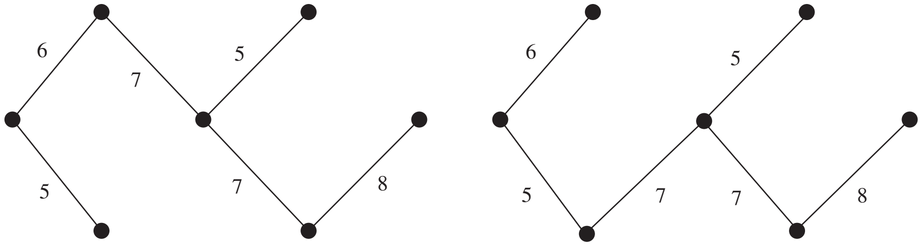
minimum spanning tree

A graph may have many spanning trees. Suppose the edges are weighted. A *minimum spanning tree* has the minimum total weights. For example, the trees in Figures 29.5b, 29.5c, 29.5d are spanning trees for the graph in Figure 29.5a. The trees in Figures 29.5c and 29.5d are minimum spanning trees.



(a)

(b) Total weight is 42



(c) Total weight is 38

(d) Total weight is 38

Figure 29.5 The trees in (c) and (d) are minimum spanning trees of the graph in (a).

The problem of finding a minimum spanning tree has many applications. Consider a company with branches in many cities. The company wants to lease telephone lines to connect all the branches together. The phone company charges different rates to connect different pairs of cities. There are many ways to connect all branches together. The cheapest way is to find a spanning tree with the minimum total rates.

29.4.1 Minimum Spanning Tree Algorithms

Prim's algorithm

How do you find a minimum spanning tree? There are several well-known algorithms for doing so. This section introduces *Prim's algorithm*. Prim's algorithm starts with a spanning tree `T` that contains an arbitrary vertex. The algorithm expands the tree by repeatedly adding a vertex with the *lowest-cost* edge incident to a vertex already in the tree. Prim's algorithm is a greedy algorithm, and it is described in Listing 29.4.

Listing 29.4 Prim's Minimum Spanning Tree Algorithm

```
Input: A connected undirected weighted G = (V, E) with nonnegative weights
Output: MST (a minimum spanning tree with vertex s as the root)

MST getMinimumSpanningTree(s) {
  Let T be a set for the vertices in the spanning tree;
```

add initial vertex

more vertices?

find a vertex

add to tree

```
  Initially, add the starting vertex, s, to T;

  while (size of T < n) {
    Find x in T and y in V – T with the smallest weight
      on the edge (x, y), as shown in Figure 29.6;
    Add y to T and set parent[y] = x;
  }
}
```

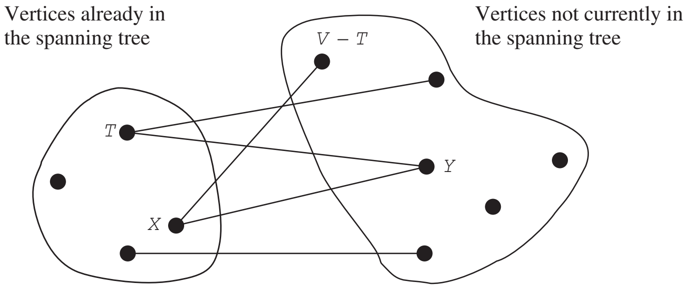

Vertices already in the spanning tree
Vertices not currently in the spanning tree
V – T
T
Y
X

FIGURE 29.6 Find a vertex `x` in `T` that connects a vertex `y` in `V – T` with the smallest weight.

example

The algorithm starts by adding the starting vertex into `T`. It then continuously adds a vertex (say `y`) from `V – T` into `T`. `y` is the vertex that is adjacent to a vertex in `T` with the smallest weight on the edge. For example, there are five edges connecting vertices in `T` and `V – T` as shown in Figure 29.6, and (`x`, `y`) is the one with the smallest weight. Consider the graph in Figure 29.7. The algorithm adds the vertices to `T` in this order:

1. Add vertex `0` to `T`.
2. Add vertex `5` to `T`, since `WeightedEdge(5, 0, 5)` has the smallest weight among all edges incident to a vertex in `T`, as shown in Figure 29.7a. The arrow line from `0` to `5` indicates that `0` is the parent of `5`.
3. Add vertex `1` to `T`, since `WeightedEdge(1, 0, 6)` has the smallest weight among all edges incident to a vertex in `T`, as shown in Figure 29.7b.
4. Add vertex `6` to `T`, since `WeightedEdge(6, 1, 7)` has the smallest weight among all edges incident to a vertex in `T`, as shown in Figure 29.7c.
5. Add vertex `2` to `T`, since `WeightedEdge(2, 6, 5)` has the smallest weight among all edges incident to a vertex in `T`, as shown in Figure 29.7d.
6. Add vertex `4` to `T`, since `WeightedEdge(4, 6, 7)` has the smallest weight among all edges incident to a vertex in `T`, as shown in Figure 29.7e.
7. Add vertex `3` to `T`, since `WeightedEdge(3, 2, 8)` has the smallest weight among all edges incident to a vertex in `T`, as shown in Figure 29.7f.

unique tree?

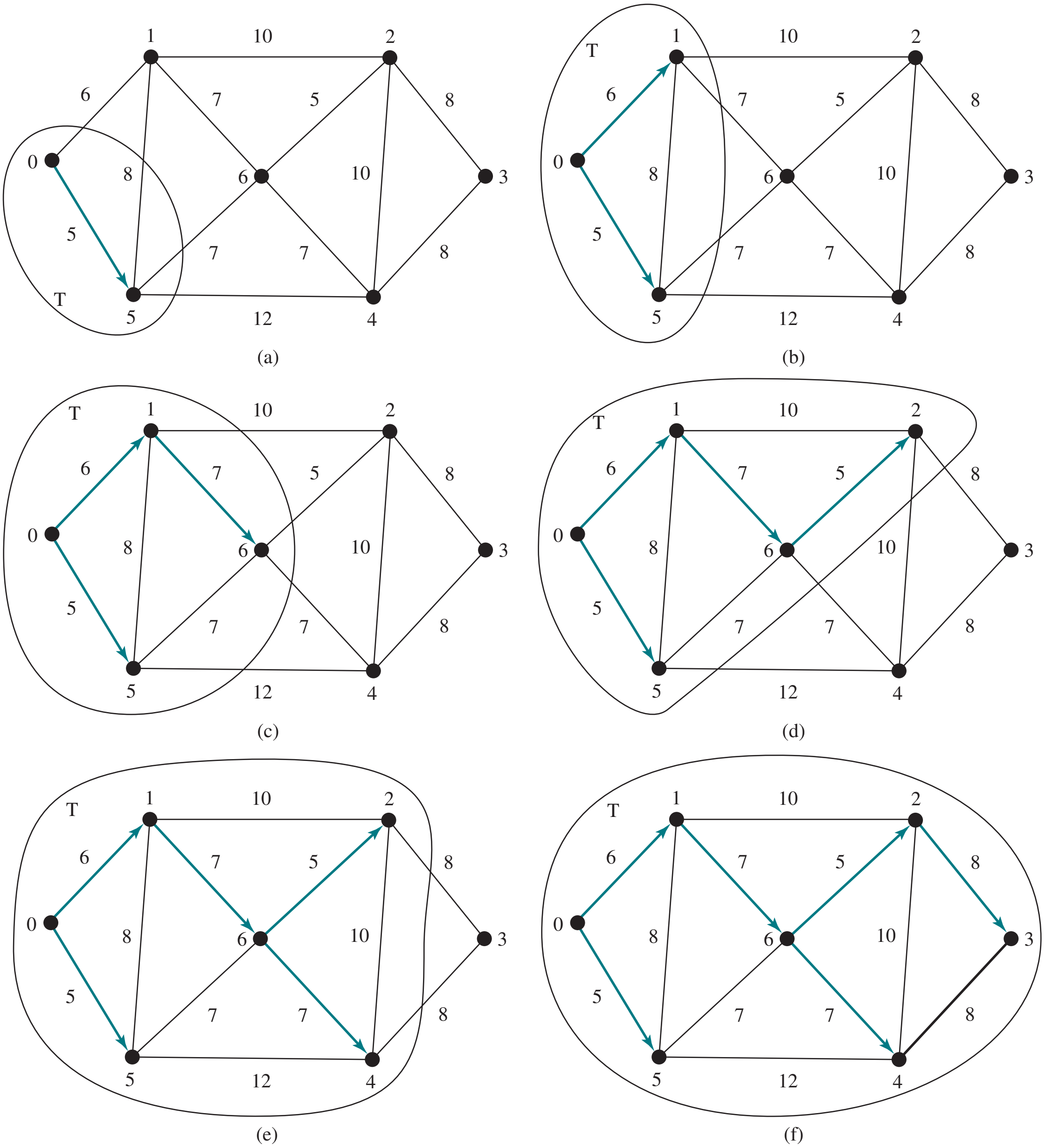
Note

A minimum spanning tree is not unique. For example, both (c) and (d) in Figure 29.5 are minimum spanning trees for the graph in Figure 29.5a. However, if the weights are distinct, the graph has a unique minimum spanning tree.

connected and undirected

Note

Assume the graph is connected and undirected. If a graph is not connected or directed, the algorithm will not work. You can modify the algorithm to find a spanning forest for any undirected graph. A spanning forest is a graph in which each connected component is a tree.


1
10
2
6
7
5
8
0
8
6
10
3
5
7
7
8
T
5
12
4
T
1
10
2
6
7
5
8
0
8
6
10
3
5
7
7
8
5
12
4
T
1
10
2
6
7
5
8
0
8
6
10
3
5
7
7
8
5
12
4
T
1
10
2
6
7
5
8
0
8
6
10
3
5
7
7
8
5
12
4
T
1
10
2
6
7
5
8
0
8
6
10
3
5
7
7
8
5
12
4
T
1
10
2
6
7
5
8
0
8
6
10
3
5
7
7
8
5
12
4

(a) (b)

(c) (d)

(e) (f)

FIGURE 29.7 The adjacent vertices with the smallest weight are added successively to **T**.

29.4.2 Refining Prim's MST Algorithm

To make it easy to identify the next vertex to add into the tree, we use **`cost[v]`** to store the cost of adding a vertex **`v`** to the spanning tree **`T`**. Initially, **`cost[s]`** is **`0`** for a starting vertex and assign infinity to **`cost[v]`** for all other vertices. The algorithm repeatedly finds a vertex **`u`** in **`V–T`** with the smallest **`cost[u]`** and moves **`u`** to **`T`**. The refined version of the algorithm is given in Listing 29.5.

LISTING 29.5 Refined Version of Prim's Algorithm

```
Input: A connected undirected weighted G = (V, E) with nonnegative weights
Output: a minimum spanning tree with the starting vertex s as the root
```

find next vertex

add a vertex to T

adjust `cost[v]`

```
MST getMinimumSpanngingTree(s) {
  Let T be a set that contains the vertices in the spanning tree;
  Initially T is empty;
  Set cost[s] = 0 and cost[v] = infinity for all other vertices in V;

  while (size of T < n) {
    Find u not in T with the smallest cost[u];
    Add u to T;
    for (each v not in T and (u, v) in E)
      if (cost[v] > w(u, v)) { // Adjust cost[v]
        cost[v] = w(u, v); parent[v] = u;
      }
  }
}
```

For an interactive demo on how the refined Prim's algorithm works, see liveexample.pearsoncmg.com/dsanimation/RefinedPrim.html.

29.4.3 Implementation of the MST Algorithm

`getMinimumSpanningTree()`

The **`getMinimumSpanningTree(int v)`** method is defined in the **`WeightedGraph`** class, as shown in Figure 29.4. It returns an instance of the **`MST`** class. The **`MST`** class is defined as an inner class in the **`WeightedGraph`** class, which extends the **`SearchTree`** class, as shown in Figure 29.8. The **`SearchTree`** class was shown in Figure 28.11. The **`MST`** class was implemented in lines 141–153 in Listing 29.2.

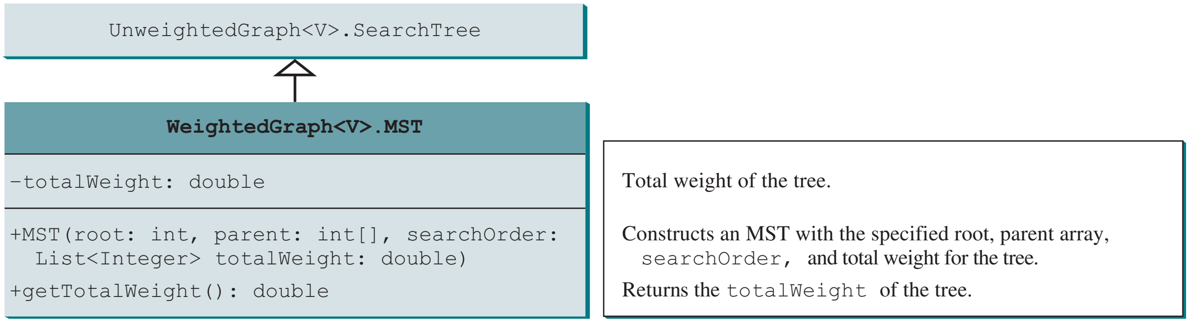

UnweightedGraph<V>.SearchTree
WeightedGraph<V>.MST
-totalWeight: double
+MST(root: int, parent: int[], searchOrder: List<Integer> totalWeight: double)
+getTotalWeight(): double
Total weight of the tree.
Constructs an MST with the specified root, parent array, searchOrder, and total weight for the tree.
Returns the totalWeight of the tree.

FIGURE 29.8 The **`MST`** class extends the **`SearchTree`** class.

The refined version of the Prim's algorithm greatly simplifies the implementation. The **`getMinimumSpanningTree`** method was implemented using the refined version of the Prim's algorithm in lines 99–138 in Listing 29.2. The **`getMinimumSpanningTree(int startingVertex)`** method sets **`cost[startingVertex]`** to **`0`** (line 105) and **`cost[v]`** to infinity for all other vertices (lines 102–104). The parent of **`startingVertex`** is set to **`-1`** (line 108). **`T`** is a list that stores the vertices added into the spanning tree (line 111). We use a list for **`T`** rather than a set in order to record the order of the vertices added to **`T`**.

Initially, **`T`** is empty. To expand **`T`**, the method performs the following operations:

1. Find the vertex **`u`** with the smallest **`cost[u]`** (lines 118–123).
2. If **`u`** is found, add it to **`T`** (line 125). Note if **`u`** is not found (**`u == -1`**), the graph is not connected. The **`break`** statement exits the while loop in this case.
3. After adding **`u`** in **`T`**, update **`cost[v]`** and **`parent[v]`** for each **`v`** adjacent to **`u`** in **`V-T`** if **`cost[v] > w(u, v)`** (lines 129–134).

After a new vertex is added to **`T`**, **`totalWeight`** is updated (line 126). Once all reachable vertices from **`s`** are added to **`T`**, an instance of **`MST`** is created (line 137). Note the method will not work if the graph is not connected. However, you can modify it to obtain a partial MST.

The **`MST`** class extends the **`SearchTree`** class (line 141). To create an instance of **`MST`**, pass **`root`**, **`parent`**, **`T`**, and **`totalWeight`** (lines 144–145). The data fields **`root`**, **`parent`**, and **`searchOrder`** are defined in the **`SearchTree`** class, which is an inner class defined in **`UnweightedGraph`**.

Note testing whether a vertex **`i`** is in **`T`** by invoking **`T.contains(i)`** takes **`O(n)`** time, since **`T`** is a list. Therefore, the overall time complexity for this implementation is $O(n^3)$. Interested readers may see Programming Exercise 29.20 for improving the implementation and reduce the complexity to $O(n^2)$.

time complexity

Listing 29.6 gives a test program that displays minimum spanning trees for the graph in Figure 29.1 and the graph in Figure 29.3a, respectively.

LISTING 29.6 `TestMinimumSpanningTree.java`

```java
public class TestMinimumSpanningTree {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    int[][] edges = {
      {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
      {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
      {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
      {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
        {3, 5, 1003},
      {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
        {4, 8, 864}, {4, 10, 496},
      {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
        {5, 6, 983}, {5, 7, 787},
      {6, 5, 983}, {6, 7, 214},
      {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
      {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
        {8, 10, 781}, {8, 11, 810},
      {9, 8, 661}, {9, 11, 1187},
      {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
      {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
    };

    WeightedGraph<String> graph1 =
      new WeightedGraph<>(vertices, edges);
    WeightedGraph<String>.MST tree1 = graph1.getMinimumSpanningTree();
    System.out.println("tree1: Total weight is " +
      tree1.getTotalWeight());
    tree1.printTree();

    edges = new int[][] {
      {0, 1, 2}, {0, 3, 8},
      {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
      {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
      {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
      {4, 2, 5}, {4, 3, 6}
    };

    WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);
    WeightedGraph<Integer>.MST tree2 =
      graph2.getMinimumSpanningTree(1);
    System.out.println("\ntree2: Total weight is " +
      tree2.getTotalWeight());
    tree2.printTree();
```

create vertices

create edges

create graph1

MST for graph1

total weight

print tree

create edges

create graph2

MST for graph2

total weight

print tree

display search order

```
47
48     System.out.println("\nShow the search order for tree1:");
49     for (int i: tree1.getSearchOrder())
50       System.out.print(graph1.getVertex(i) + " ");
51   }
52 }
```

```
Total weight is 6513.0
Root is: Seattle
Edges: (Seattle, San Francisco) (San Francisco, Los Angeles)
  (Los Angeles, Denver) (Denver, Kansas City) (Kansas City, Chicago)
  (New York, Boston) (Chicago, New York) (Dallas, Atlanta)
  (Atlanta, Miami) (Kansas City, Dallas) (Dallas, Houston)

Total weight is 14.0
Root is: 1
Edges: (1, 0) (3, 2) (1, 3) (2, 4)

Show the search order for tree1:
Seattle San Francisco Los Angeles Denver Kansas City Dallas

Houston Chicago Atlanta Miami New York Boston
```

The program creates a weighted graph for Figure 29.1 in line 27. It then invokes `getMinimumSpanningTree()` (line 28) to return an `MST` that represents a minimum spanning tree for the graph. Invoking `printTree()` (line 31) on the `MST` object displays the edges in the tree. Note that `MST` is a subclass of `Tree`. The `printTree()` method is defined in the `SearchTree` class.

graphical illustration

The graphical illustration of the minimum spanning tree is shown in Figure 29.9. The vertices are added to the tree in this order: Seattle, San Francisco, Los Angeles, Denver, Kansas City, Dallas, Houston, Chicago, Atlanta, Miami, New York, and Boston.

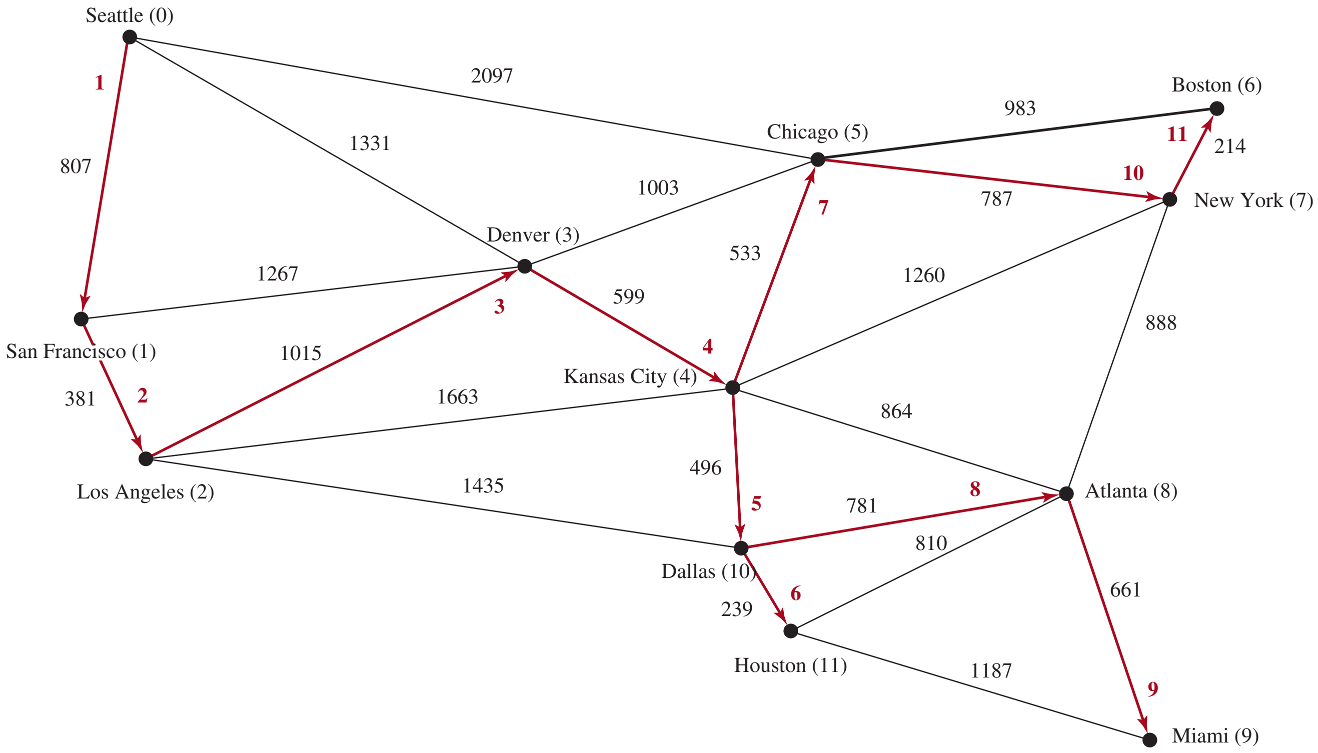
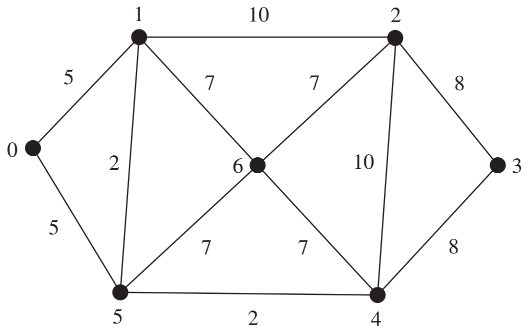

Seattle (0)
1
807
2097
1331
Boston (6)
983
11
214
Chicago (5)
10
New York (7)
1003
787
7
Denver (3)
533
1260
1267
599
3
San Francisco (1)
1015
4
888
Kansas City (4)
381
2
1663
864
Los Angeles (2)
1435
496
5
781
8
Atlanta (8)
810
Dallas (10)
6
661
239
Houston (11)
1187
9
Miami (9)

FIGURE 29.9 The edges in a minimum spanning tree for the cities are highlighted.

Check Point

29.4.1 Find a minimum spanning tree for the following graph:



29.4.2 Is a minimum spanning tree unique if all edges have different weights?

29.4.3 If you use an adjacency matrix to represent weighted edges, what will be the time complexity for Prim's algorithm?

29.4.4 What happens to the `getMinimumSpanningTree()` method in `WeightedGraph` if the graph is not connected? Verify your answer by writing a test program that creates an unconnected graph and invokes the `getMinimumSpanningTree()` method. How do you fix the problem by obtaining a partial MST?

29.4.5 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    WeightedGraph<Character> graph = new WeightedGraph<>();
    graph.addVertex('U');
    graph.addVertex('V');
    graph.addVertex('X');
    int indexForU = graph.getIndex('U');
    int indexForV = graph.getIndex('V');
    int indexForX = graph.getIndex('X');
    System.out.println("indexForU is " + indexForU);
    System.out.println("indexForV is " + indexForV);
    System.out.println("indexForX is " + indexForV);
    graph.addEdge(indexForU, indexForV, 3.5);
    graph.addEdge(indexForV, indexForU, 3.5);
    graph.addEdge(indexForU, indexForX, 2.1);
    graph.addEdge(indexForX, indexForU, 2.1);
    graph.addEdge(indexForV, indexForX, 3.1);
    graph.addEdge(indexForX, indexForV, 3.1);
    WeightedGraph<Character>.MST mst
      = graph.getMinimumSpanningTree();
    graph.printWeightedEdges();
    System.out.println(mst.getTotalWeight());
    mst.printTree();
  }
}
```

29.5 Finding Shortest Paths

The shortest path between two vertices is a path with the minimum total weights.

Key Point

Dijkstra's algorithm

Given a graph with nonnegative weights on the edges, a well-known algorithm for finding a *shortest path* between two vertices was discovered by Edsger Dijkstra, a Dutch computer scientist. In order to find a shortest path from vertex `s` to vertex `v`, *Dijkstra's algorithm* finds the shortest path from `s` to all vertices. So Dijkstra's algorithm is known as a

single-source shortest path
shortest path

single-source shortest-path algorithm. The algorithm uses **`cost[v]`** to store the cost of a *shortest path* from vertex **`v`** to the source vertex **`s`**. **`cost[s]`** is **`0`**. Initially assign infinity to **`cost[v]`** for all other vertices. The algorithm repeatedly finds a vertex **`u`** in **`V–T`** with the smallest **`cost[u]`** and moves **`u`** to **`T`**.

The algorithm is described in Listing 29.7.

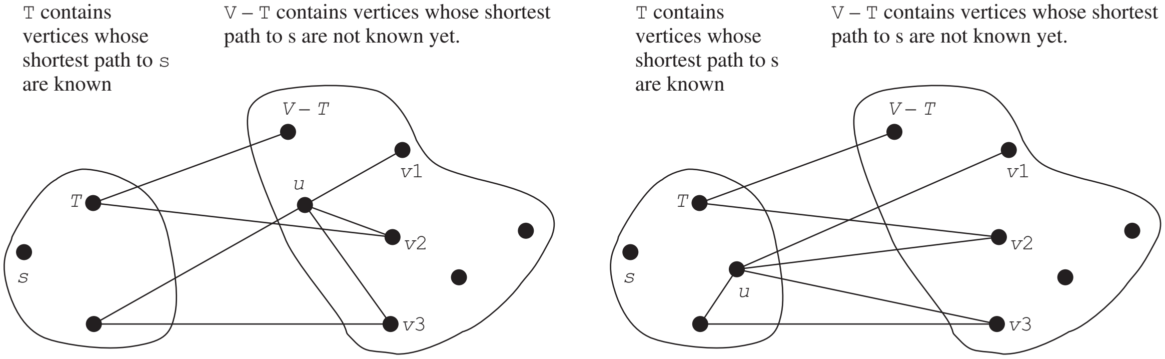
LISTING 29.7 Dijkstra's Single-Source Shortest-Path Algorithm

```
Input: a graph G = (V, E) with nonnegative weights
Output: a shortest-path tree with the source vertex s as the root

 1  ShortestPathTree getShortestPath(s) {
 2    Let T be a set that contains the vertices whose
 3      paths to s are known; Initially T is empty;
 4    Set cost[s] = 0; and cost[v] = infinity for all other vertices in V;
 5
 6    while (size of T < n) {
 7      Find u not in T with the smallest cost[u];
 8      Add u to T;
 9      for (each v not in T and (u, v) in E)
10        if (cost[v] > cost[u] + w(u, v)) {
11          cost[v] = cost[u] + w(u, v); parent[v] = u;
12        }
13    }
14  }
```

find next vertex
add a vertex to T
adjust `cost[v]`

This algorithm is very similar to Prim's for finding a minimum spanning tree. Both algorithms divide the vertices into two sets: **`T`** and **`V - T`**. In the case of Prim's algorithm, set **`T`** contains the vertices that are already added to the tree. In the case of Dijkstra's, set **`T`** contains the vertices whose shortest paths to the source have been found. Both algorithms repeatedly find a vertex from **`V - T`** and add it to **`T`**. In the case of Prim's algorithm, the vertex is adjacent to some vertex in the set with the minimum weight on the edge. In Dijkstra's algorithm, the vertex is adjacent to some vertex in the set with the minimum total cost to the source.

The algorithm starts by setting **`cost[s]`** to **`0`** (line 4), sets **`cost[v]`** to infinity for all other vertices. It then continuously adds a vertex (say **`u`**) from **`V–T`** into **`T`** with smallest **`cost[u]`** (lines 7–8), as shown in Figure 29.10a. After adding **`u`** to **`T`**, the algorithm updates **`cost[v]`** and **`parent[v]`** for each **`v`** not in **`T`** if **`(u, v)`** is in **`T`** and **`cost[v] > cost[u] + w(u, v)`** (lines 10–12).

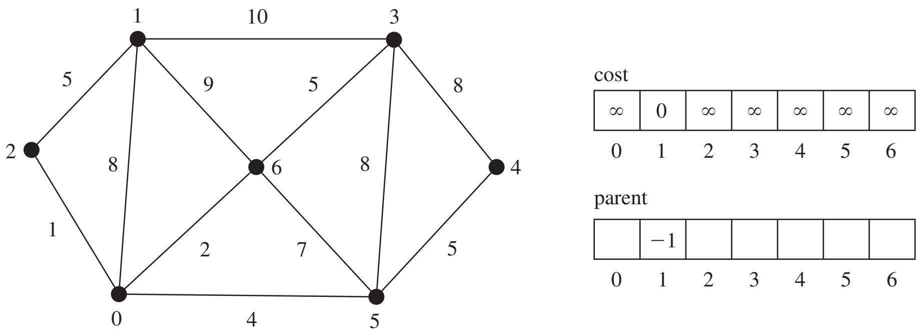

T contains vertices whose shortest path to s are known
V – T contains vertices whose shortest path to s are not known yet.
V – T
T
s
u
v1
v2
v3
T contains vertices whose shortest path to s are known
V – T contains vertices whose shortest path to s are not known yet.
V – T
T
s
u
v1
v2
v3

(a) Before moving **`u`** to **`T`**

(b) After moving **`u`** to **`T`**

FIGURE 29.10 (a) Find a vertex **`u`** in **`V–T`** with the smallest **`cost[u]`**. (b) Update **`cost[v]`** for v in **`V–T`** and **`v`** is adjacent to **`u`**.

Let us illustrate Dijkstra's algorithm using the graph in Figure 29.11a. Suppose the source vertex is `1`. Therefore, `cost[1] = 0` and the costs for all other vertices are initially ∞, as shown in Figure 29.11b. We use the `parent[i]` to denote the parent of `i` in the path. For convenience, set the parent of the source node to `−1`.



cost

∞	0	∞	∞	∞	∞	∞
0	1	2	3	4	5	6

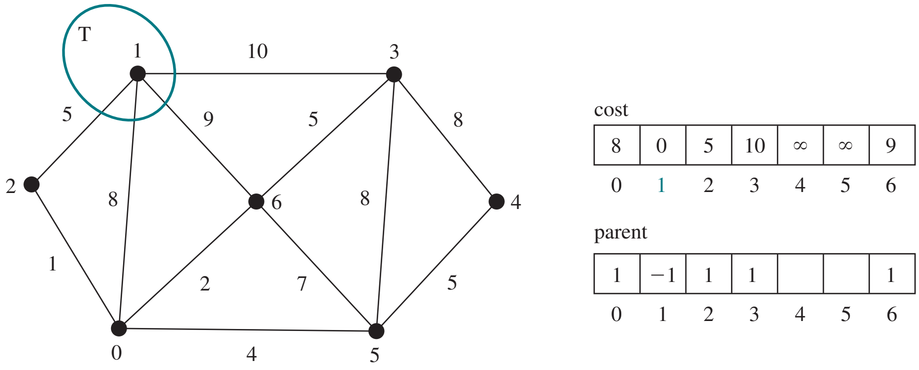
parent

	−1					
0	1	2	3	4	5	6

(a) (b)

FIGURE 29.11 The algorithm will find all shortest paths from source vertex `1`.

Initially set `T` is empty. The algorithm selects the vertex with the smallest cost. In this case, the vertex is `1`. The algorithm adds `1` to `T`, as shown in Figure 29.12a. Afterward, it adjusts the cost for each vertex adjacent to `1`. The cost for vertices 2, 0, 6, and 3 and their parents are now updated, as shown in Figure 29.12b.



cost

8	0	5	10	∞	∞	9
0	1	2	3	4	5	6

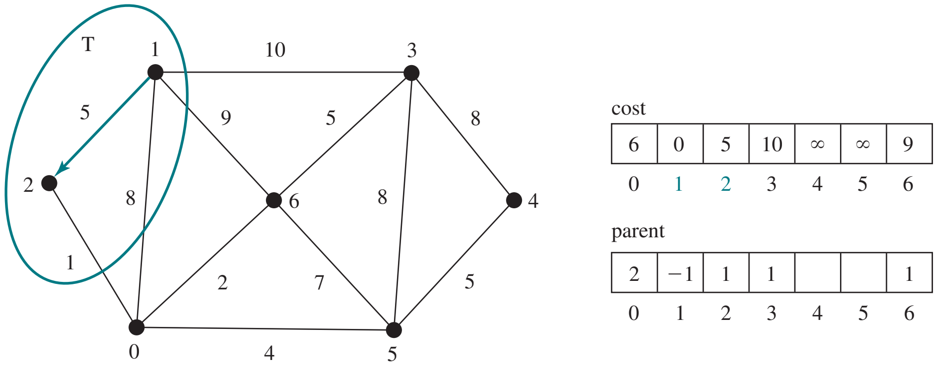
parent

1	−1	1	1			1
0	1	2	3	4	5	6

(a) (b)

FIGURE 29.12 Now vertex `1` is in set `T`.

Vertices `2`, `0`, `6`, and `3` are adjacent to the source vertex, and vertex `2` is the one in `V−T` with the smallest cost, so add `2` to `T`, as shown in Figure 29.13 and update the cost and parent for vertices in `V−T` and adjacent to `2`. `cost[0]` is now updated to `6` and its parent is set to `2`. The arrow line from `1` to `2` indicates `1` is the parent of `2` after `2` is added into `T`.



(a)

cost

0	1	2	3	4	5	6
6	0	5	10	∞	∞	9

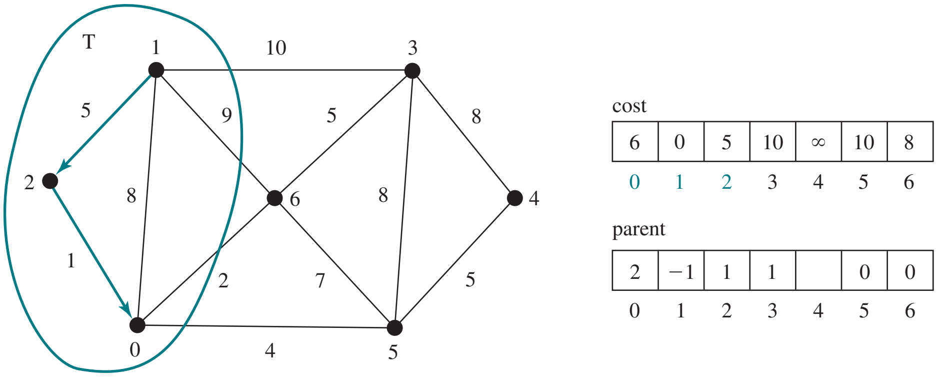
parent

0	1	2	3	4	5	6
2	−1	1	1			1

(b)

FIGURE 29.13 Now vertices `1` and `2` are in set `T`.

Now `T` contains {`1`, `2`}. Vertex `0` is the one in `V-T` with the smallest cost, so add `0` to `T`, as shown in Figure 29.14 and update the cost and parent for vertices in `V-T` and adjacent to `0` if applicable. `cost[5]` is now updated to `10` and its parent is set to `0` and `cost[6]` is now updated to `8` and its parent is set to `0`.



(a)

cost

0	1	2	3	4	5	6
6	0	5	10	∞	10	8

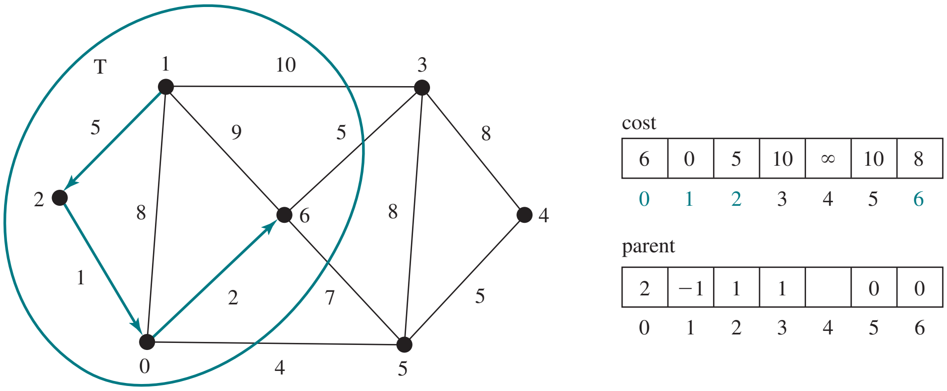
parent

0	1	2	3	4	5	6
2	−1	1	1		0	0

(b)

FIGURE 29.14 Now vertices {`1`, `2`, `0`} are in set `T`.

Now `T` contains {`1`, `2`, `0`}. Vertex `6` is the one in `V-T` with the smallest cost, so add `6` to `T`, as shown in Figure 29.15 and update the cost and parent for vertices in `V-T` and adjacent to `6` if applicable.



(a)

cost

0	1	2	3	4	5	6
6	0	5	10	∞	10	8

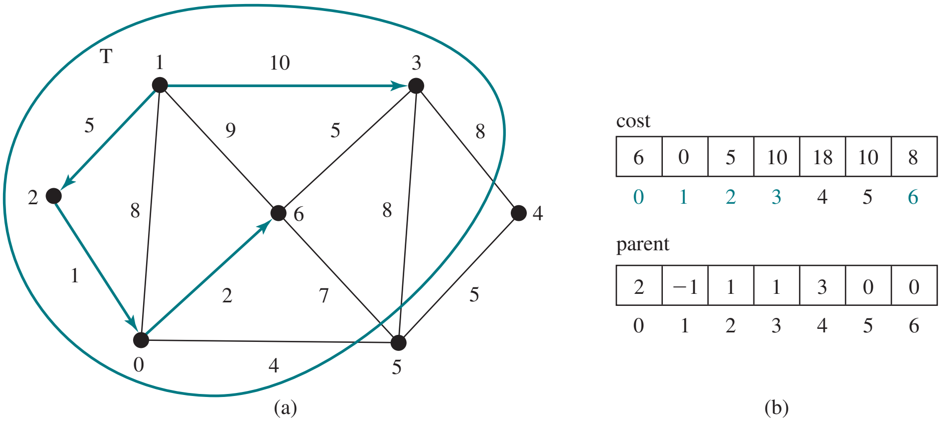
parent

0	1	2	3	4	5	6
2	−1	1	1		0	0

(b)

FIGURE 29.15 Now vertices {`1`, `2`, `0`, `6`} are in set `T`.

Now **T** contains {**1**, **2**, **0**, **6**}. Vertex **3** or **5** is the one in **V–T** with the smallest cost. You may add either **3** or **5** into **T**. Let us add **3** to **T**, as shown in Figure 29.16 and update the cost and parent for vertices in **V–T** and adjacent to **3** if applicable. `cost[4]` is now updated to **18** and its parent is set to **3**.



cost

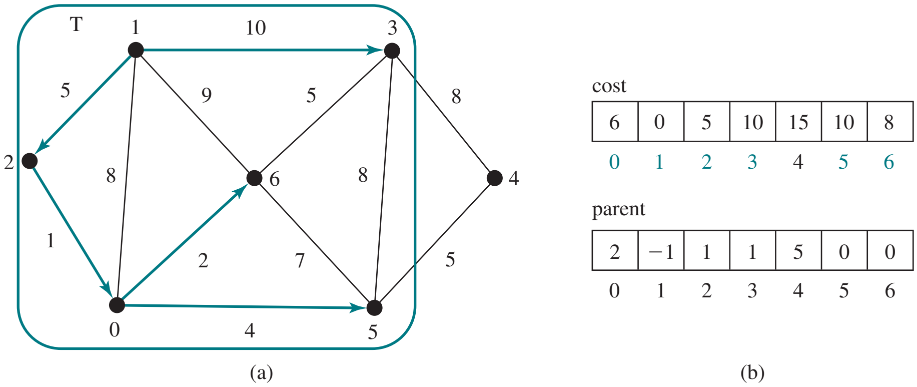
6	0	5	10	18	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1	3	0	0
0	1	2	3	4	5	6

FIGURE 29.16 Now vertices {**1**, **2**, **0**, **6**, **3**} are in set **T**.

Now **T** contains {**1**, **2**, **0**, **6**, **3**}. Vertex **5** is the one in **V–T** with the smallest cost, so add **5** to **T**, as shown in Figure 29.17 and update the cost and parent for vertices in **V–T** and adjacent to **5** if applicable. `cost[4]` is now updated to **15**, and its parent is set to **5**.



cost

6	0	5	10	15	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1	5	0	0
0	1	2	3	4	5	6

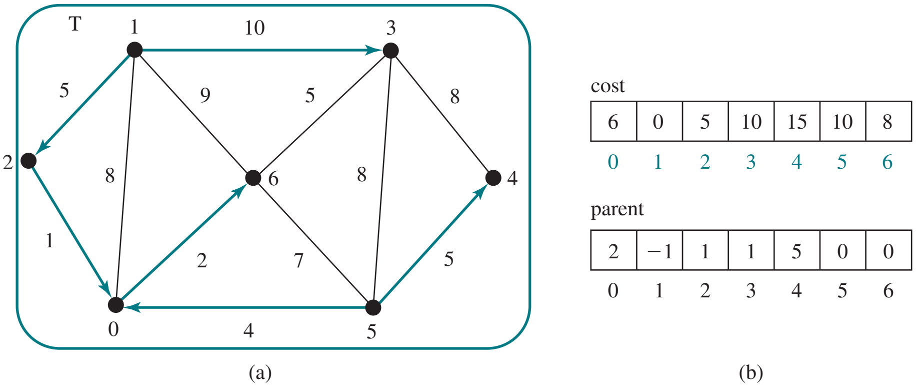
FIGURE 29.17 Now vertices {**1**, **2**, **0**, **6**, **3**, **5**} are in set **T**. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Now **T** contains {**1**, **2**, **0**, **6**, **3**, **5**}. Vertex **4** is the one in **V–T** with the smallest cost, so add **4** to **T**, as shown in Figure 29.18.

shortest-path tree

As you can see, the algorithm essentially finds all shortest paths from a source vertex, which produces a tree rooted at the source vertex. We call this tree a *single-source all-shortest-path tree* (or simply a *shortest-path tree*). To model this tree, define a class named `ShortestPathTree` that extends the `SearchTree` class, as shown in Figure 29.19. `ShortestPathTree` is defined as an inner class in `WeightedGraph` in lines 200–224 in Listing 29.2.

The `getShortestPath(int sourceVertex)` method was implemented in lines 156–197 in Listing 29.2. The method sets `cost[sourceVertex]` to `0` (line 162) and `cost[v]` to



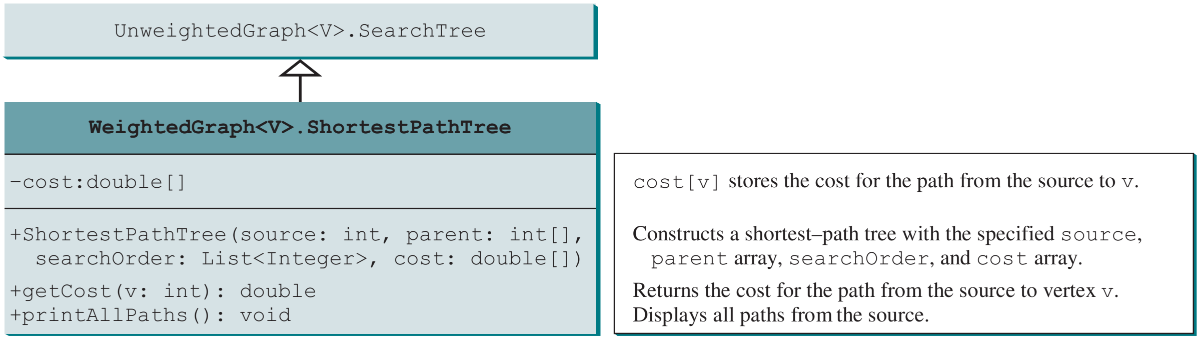
cost

6	0	5	10	15	10	8
0	1	2	3	4	5	6

parent

2	−1	1	1	5	0	0
0	1	2	3	4	5	6

FIGURE 29.18 Now vertices {`1`, `2`, `6`, `0`, `3`, `5`, `4`} are in set `T`. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.


UnweightedGraph<V>.SearchTree
WeightedGraph<V>.ShortestPathTree
-cost:double[]
+ShortestPathTree(source: int, parent: int[], searchOrder: List<Integer>, cost: double[])
+getCost(v: int): double
+printAllPaths(): void

`cost[v]` stores the cost for the path from the source to `v`.

Constructs a shortest–path tree with the specified `source`, `parent` array, `searchOrder`, and `cost` array.

Returns the cost for the path from the source to vertex `v`.
Displays all paths from the source.

FIGURE 29.19 `WeightedGraph<V>.ShortestPathTree` extends `UnweightedGraph<V>.SearchTree`. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

infinity for all other vertices (lines 159–161). The parent of `sourceVertex` is set to `-1` (line 166). `T` is a list that stores the vertices added into the shortest-path tree (line 169). We use a list for `T` rather than a set in order to record the order of the vertices added to `T`.

Initially, `T` is empty. To expand `T`, the method performs the following operations:

1. Find the vertex `u` with the smallest `cost[u]` (lines 175–181).
2. If `u` is found, add it to `T` (line 183). Note that if `u` is not found (`u == -1`), the graph is not connected. The `break` statement exits the while loop in this case.
3. After adding `u` in `T`, update `cost[v]` and `parent[v]` for each `v` adjacent to `u` in `V-T` if `cost[v] > cost[u] + w(u, v)` (lines 186–192).

`ShortestPathTree` class

Once all vertices from `V` are added to `T`, an instance of `ShortestPathTree` is created (line 196).

The `ShortestPathTree` class extends the `SearchTree` class (line 200). To create an instance of `ShortestPathTree`, pass `sourceVertex`, `parent`, `T`, and `cost` (lines 204–205). `sourceVertex` becomes the root in the tree. The data fields `root`, `parent`, and `searchOrder` are defined in the `SearchTree` class, which is an inner class defined in `UnweightedGraph`.

Dijkstra's algorithm time complexity

Note testing whether a vertex `i` is in `T` by invoking `T.conatins(i)` takes `O(n)` time, since `T` is a list. Therefore, the overall time complexity for this implementation is $O(n^3)$. Interested readers may see Programming Exercise 29.20 for improving the implementation and reducing the complexity to $O(n^2)$.

greedy and dynamic programming

Dijkstra's algorithm is a combination of a greedy algorithm and dynamic programming. It is a greedy algorithm in the sense that it always adds a new vertex that has the shortest distance to the source. It stores the shortest distance of each known vertex to the source, and uses it later to avoid redundant computing, so Dijkstra's algorithm also uses dynamic programming.

Listing 29.8 gives a test program that displays the shortest paths from Chicago to all other cities in Figure 29.1, and the shortest paths from vertex `3` to all vertices for the graph in Figure 29.3a, respectively.

LISTING 29.8 TestShortestPath.java

```java
public class TestShortestPath {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    int[][] edges = {
      {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
      {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
      {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
      {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
        {3, 5, 1003},
      {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
        {4, 8, 864}, {4, 10, 496},
      {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
        {5, 6, 983}, {5, 7, 787},
      {6, 5, 983}, {6, 7, 214},
      {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
      {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
        {8, 10, 781}, {8, 11, 810},
      {9, 8, 661}, {9, 11, 1187},
      {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
      {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
    };

    WeightedGraph<String> graph1 =
      new WeightedGraph<>(vertices, edges);
    WeightedGraph<String>.ShortestPathTree tree1 =
      graph1.getShortestPath(graph1.getIndex("Chicago"));
    tree1.printAllPaths();

    // Display shortest paths from Houston to Chicago
    System.out.print("Shortest path from Houston to Chicago: ");
    java.util.List<String> path
      = tree1.getPath(graph1.getIndex("Houston"));
    for (String s: path) {
      System.out.print(s + " ");
    }

    edges = new int[][] {
      {0, 1, 2}, {0, 3, 8},
      {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
      {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
      {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
      {4, 2, 5}, {4, 3, 6}
    };
    WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);
    WeightedGraph<Integer>.ShortestPathTree tree2 =
      graph2.getShortestPath(3);
    System.out.println("\n");
    tree2.printAllPaths();
  }
}
```

vertices

edges

create graph1

shortest path

create edges

create graph2

print paths

```
All shortest paths from Chicago are:
A path from Chicago to Seattle: Chicago Seattle (cost: 2097.0)
A path from Chicago to San Francisco: Chicago Denver San Francisco
  (cost: 2270.0)
A path from Chicago to Los Angeles: Chicago Denver Los Angeles
  (cost: 2018.0)
A path from Chicago to Denver: Chicago Denver (cost: 1003.0)
A path from Chicago to Kansas City: Chicago Kansas City (cost: 533.0)
A path from Chicago to Chicago: Chicago (cost: 0.0)
A path from Chicago to Boston: Chicago Boston (cost: 983.0)
A path from Chicago to New York: Chicago New York (cost: 787.0)
A path from Chicago to Atlanta: Chicago Kansas City Atlanta
  (cost: 1397.0)
A path from Chicago to Miami:
  Chicago Kansas City Atlanta Miami (cost: 2058.0)
A path from Chicago to Dallas:
  Chicago Kansas City Dallas (cost: 1029.0)
A path from Chicago to Houston:
  Chicago Kansas City Dallas Houston (cost: 1268.0)
Shortest path from Houston to Chicago:
  Houston Dallas Kansas City Chicago

All shortest paths from 3 are:
A path from 3 to 0: 3 1 0 (cost: 5.0)
A path from 3 to 1: 3 1 (cost: 3.0)
A path from 3 to 2: 3 2 (cost: 4.0)
A path from 3 to 3: 3 (cost: 0.0)
A path from 3 to 4: 3 4 (cost: 6.0)
```

The program creates a weighted graph for Figure 29.1 in line 27. It then invokes the `getShortestPath(graph1.getIndex("Chicago"))` method to return a `Path` object that contains all shortest paths from Chicago. Invoking `printAllPaths()` on the `ShortestPathTree` object displays all the paths (line 30).

The graphical illustration of all shortest paths from Chicago is shown in Figure 29.20. The shortest paths from Chicago to the cities are found in this order: Kansas City, New York, Boston, Denver, Dallas, Houston, Atlanta, Los Angeles, Miami, Seattle, and San Francisco.

Check Point

29.5.1 Trace Dijkstra's algorithm for finding shortest paths from Boston to all other cities in Figure 29.1.

29.5.2 Is a shortest path between two vertices unique if all edges have different weights?

29.5.3 If you use an adjacency matrix to represent weighted edges, what would be the time complexity for Dijkstra's algorithm?

29.5.4 What happens to the `getShortestPath()` method in `WeightedGraph` if the source vertex cannot reach all vertices in the graph? Verify your answer by writing a test program that creates an unconnected graph and invoke the `getShortestPath()` method. How do you fix the problem by obtaining a partial shortest-path tree?

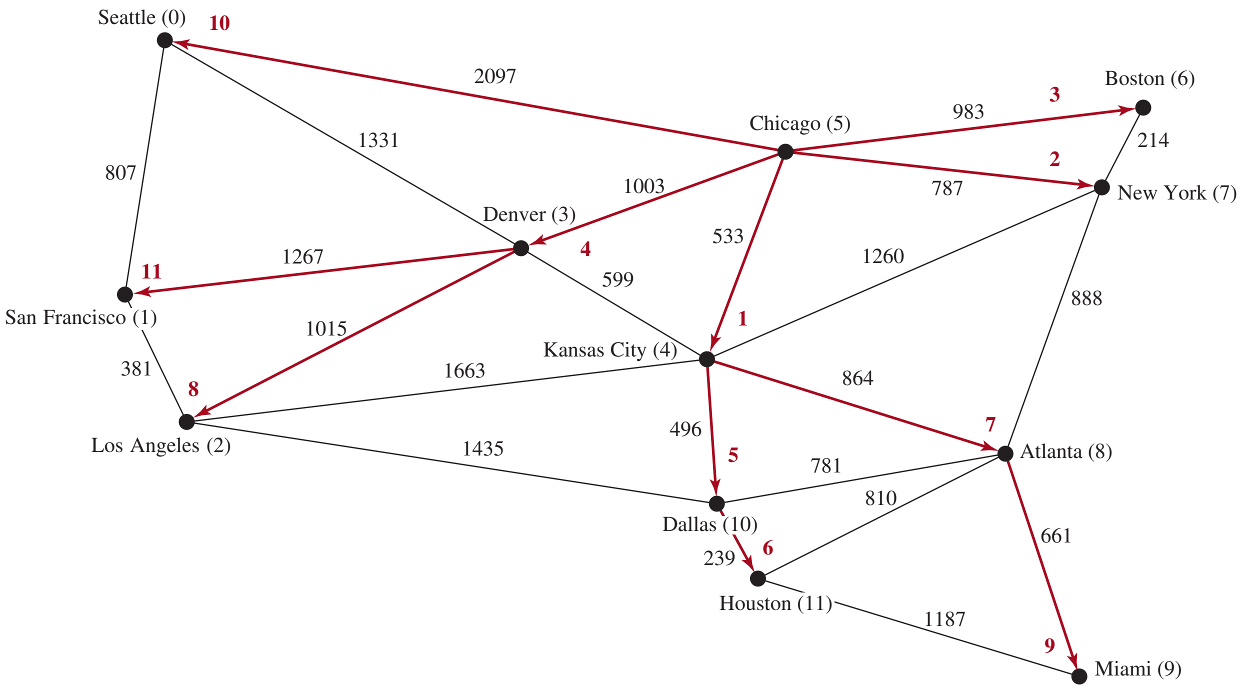

Seattle (0)
10
2097
Boston (6)
3
983
Chicago (5)
214
1331
2
807
787
New York (7)
1003
Denver (3)
4
533
1267
11
1260
599
888
San Francisco (1)
1015
1
Kansas City (4)
381
1663
864
8
Los Angeles (2)
496
7
Atlanta (8)
1435
5
781
810
Dallas (10)
661
239
6
Houston (11)
1187
9
Miami (9)

FIGURE 29.20 The shortest paths from Chicago to all other cities are highlighted.

29.5.5 If there is no path from vertex `v` to the source vertex, what will be `cost[v]`?

29.5.6 Assume the graph is connected; will the `getShortestPath` method find the shortest paths correctly if lines 159–161 in `WeightedGraph` are deleted?

29.5.7 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    WeightedGraph<Character> graph = new WeightedGraph<>();
    graph.addVertex('U');
    graph.addVertex('V');
    graph.addVertex('X');
    int indexForU = graph.getIndex('U');
    int indexForV = graph.getIndex('V');
    int indexForX = graph.getIndex('X');
    System.out.println("indexForU is " + indexForU);
    System.out.println("indexForV is " + indexForV);
    System.out.println("indexForX is " + indexForV);
    graph.addEdge(indexForU, indexForV, 3.5);
    graph.addEdge(indexForV, indexForU, 3.5);
    graph.addEdge(indexForU, indexForX, 2.1);
    graph.addEdge(indexForX, indexForU, 2.1);
    graph.addEdge(indexForV, indexForX, 3.1);
    graph.addEdge(indexForX, indexForV, 3.1);
    WeightedGraph<Character>.ShortestPathTree tree =
      graph.getShortestPath(1);
    graph.printWeightedEdges();
    tree.printTree();
  }
}
```

29.6 Case Study: The Weighted Nine Tails Problem

Key Point

The weighted nine tails problem can be reduced to the weighted shortest path problem.

Section 28.10 presented the nine tails problem and solved it using the BFS algorithm. This section presents a variation of the nine tails problem and solves it using the shortest-path algorithm.

The nine tails problem is to find the minimum number of the moves that lead to all coins facing down. Each move flips a head coin and its neighbors. The weighted nine tails problem assigns the number of flips as a weight on each move. For example, you can move from the coins in Figure 29.21a to those in Figure 29.21b by flipping the first coin in the first row and its two neighbors. Thus, the weight for this move is `3`. You can move from the coins in Figure 29.21c to Figure 29.21d by flipping the five coins. So the weight for this move is `5`.

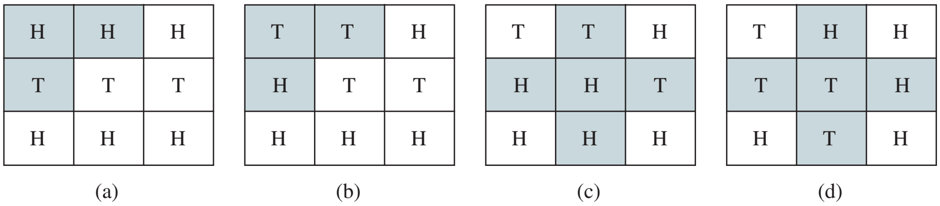


FIGURE 29.21 The weight for each move is the number of flips for the move.

The weighted nine tails problem can be reduced to finding a shortest path from a starting node to the target node in an edge-weighted graph. The graph has 512 nodes. Create an edge from node `v` to `u` if there is a move from node `u` to node `v`. Assign the number of flips to be the weight of the edge.

Recall in Section 28.10, we defined a class `NineTailModel` for modeling the nine tails problem. We now define a new class named `WeightedNineTailModel` that extends `NineTailModel`, as shown in Figure 29.22.

The `NineTailModel` class creates a `Graph` and obtains a `Tree` rooted at the target node `511`. `WeightedNineTailModel` is the same as `NineTailModel` except that it creates a `WeightedGraph` and obtains a `ShortestPathTree` rooted at the target node `511`. The method `getEdges()` finds all edges in the graph. The `getNumberOfFlips(int u, int v)` method returns the number of flips from node `u` to node `v`. The `getNumberOfFlips(int u)` method returns the number of flips from node `u` to the target node.

Listing 29.9 implements the `WeightedNineTailModel`.

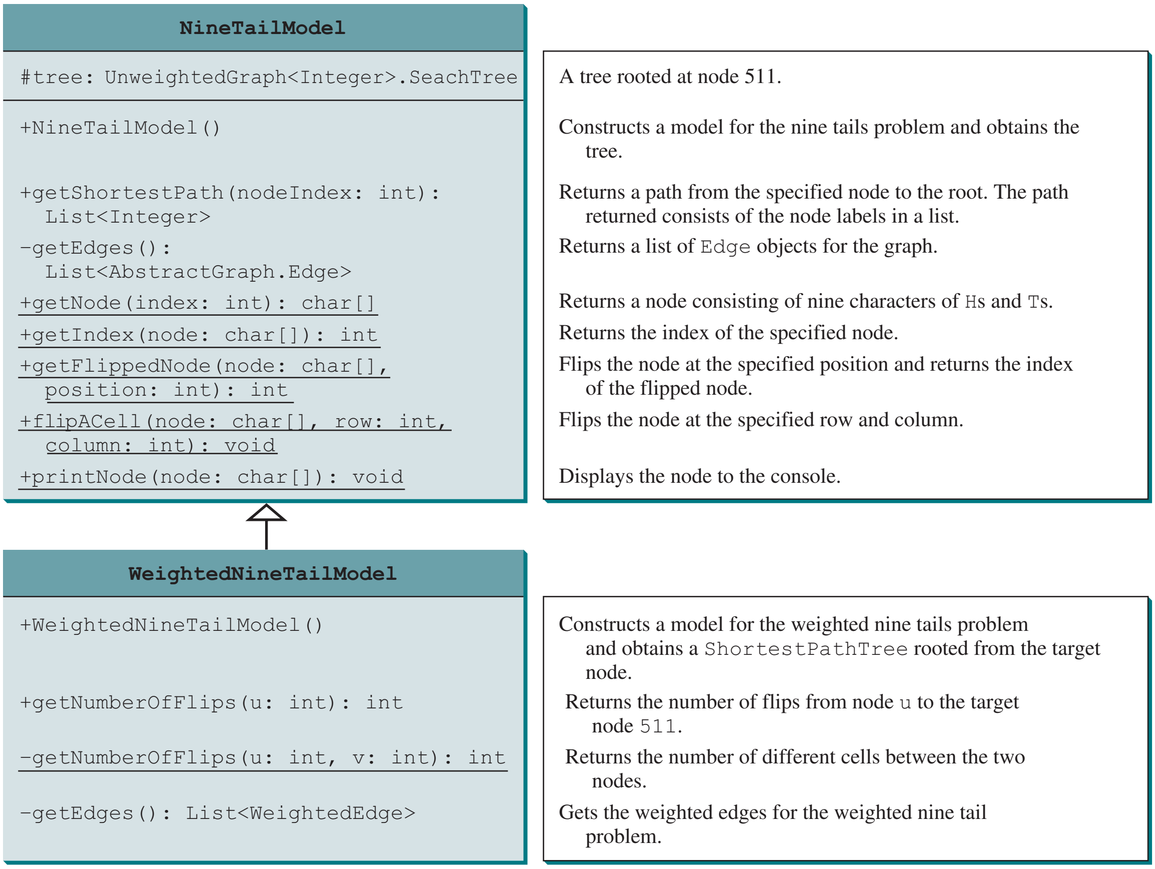

NineTailModel
#tree: UnweightedGraph<Integer>.SeachTree
+NineTailModel()
+getShortestPath(nodeIndex: int): List<Integer>
-getEdges(): List<AbstractGraph.Edge>
+getNode(index: int): char[]
+getIndex(node: char[]): int
+getFlippedNode(node: char[], position: int): int
+flipACell(node: char[], row: int, column: int): void
+printNode(node: char[]): void
A tree rooted at node 511.
Constructs a model for the nine tails problem and obtains the tree.
Returns a path from the specified node to the root. The path returned consists of the node labels in a list.
Returns a list of Edge objects for the graph.
Returns a node consisting of nine characters of Hs and Ts.
Returns the index of the specified node.
Flips the node at the specified position and returns the index of the flipped node.
Flips the node at the specified row and column.
Displays the node to the console.
WeightedNineTailModel
+WeightedNineTailModel()
+getNumberOfFlips(u: int): int
-getNumberOfFlips(u: int, v: int): int
-getEdges(): List<WeightedEdge>
Constructs a model for the weighted nine tails problem and obtains a ShortestPathTree rooted from the target node.
Returns the number of flips from node u to the target node 511.
Returns the number of different cells between the two nodes.
Gets the weighted edges for the weighted nine tail problem.

Figure 29.22 The `WeightedNineTailModel` class extends `NineTailModel`.

Listing 29.9 WeightedNineTailModel.java

```java
import java.util.*;

public class WeightedNineTailModel extends NineTailModel {
  /** Construct a model */
  public WeightedNineTailModel() {
    // Create edges
    List<WeightedEdge> edges = getEdges();

    // Create a graph
    WeightedGraph<Integer> graph = new WeightedGraph<Integer>(
      edges, NUMBER_OF_NODES);

    // Obtain a shortest-path tree rooted at the target node
    tree = graph.getShortestPath(511);
  }

  /** Create all edges for the graph */
  private List<WeightedEdge> getEdges() {
    // Store edges
    List<WeightedEdge> edges = new ArrayList<>();
```

extends `NineTailModel`

constructor

get edges

create a graph

get a tree

get weighted edges

get adjacent node

weight

add an edge

number of flips

total number of flips

```
21
22      for (int u = 0; u < NUMBER_OF_NODES; u++) {
23        for (int k = 0; k < 9; k++) {
24          char[] node = getNode(u); // Get the node for vertex u
25          if (node[k] == 'H') {
26            int v = getFlippedNode(node, k);
27            int numberOfFlips = getNumberOfFlips(u, v);
28
29            // Add edge (v, u) for a legal move from node u to node v
30            edges.add(new WeightedEdge(v, u, numberOfFlips));
31          }
32        }
33      }
34
35      return edges;
36    }
37
38    private static int getNumberOfFlips(int u, int v) {
39      char[] node1 = getNode(u);
40      char[] node2 = getNode(v);
41
42      int count = 0; // Count the number of different cells
43      for (int i = 0; i < node1.length; i++)
44        if (node1[i] != node2[i]) count++;
45
46      return count;
47    }
48
49    public int getNumberOfFlips(int u) {
50      return (int)((WeightedGraph<Integer>.ShortestPathTree)tree)
51        .getCost(u);
52    }
53  }
```

`WeightedNineTailModel` extends `NineTailModel` to build a `WeightedGraph` to model the weighted nine tails problem (lines 10–11). For each node `u`, the `getEdges()` method finds a flipped node `v` and assigns the number of flips as the weight for edge (`v`, `u`) (line 30). The `getNumberOfFlips(int u, int v)` method returns the number of flips from node `u` to node `v` (lines 38–47). The number of flips is the number of the different cells between the two nodes (line 44).

The `WeightedNineTailModel` obtains a `ShortestPathTree` rooted at the target node `511` (line 14). Note `tree` is a protected data field defined in `NineTailModel` and `ShortestPathTree` is a subclass of `Tree`. The methods defined in `NineTailModel` use the `tree` property.

The `getNumberOfFlips(int u)` method (lines 49–52) returns the number of flips from node `u` to the target node, which is the cost of the path from node `u` to the target node. This cost can be obtained by invoking the `getCost(u)` method defined in the `ShortestPathTree` class (line 51).

Listing 29.10 gives a program that prompts the user to enter an initial node and displays the minimum number of flips to reach the target node.

LISTING 29.10 WeightedNineTail.java

```
import java.util.Scanner;

public class WeightedNineTail {
  public static void main(String[] args) {
    // Prompt the user to enter the nine coins' Hs and Ts
```

```
 6      System.out.print("Enter an initial nine coins' Hs and Ts: ");
 7      Scanner input = new Scanner(System.in);
 8      String s = input.nextLine();
 9      char[] initialNode = s.toCharArray();
10
11      WeightedNineTailModel model = new WeightedNineTailModel();
12      java.util.List<Integer> path =
13        model.getShortestPath(NineTailModel.getIndex(initialNode));
14
15      System.out.println("The steps to flip the coins are ");
16      for (int i = 0; i < path.size(); i++)
17        NineTailModel.printNode(NineTailModel.getNode(path.get(i)));
18
19      System.out.println("The number of flips is " +
20        model.getNumberOfFlips(NineTailModel.getIndex(initialNode)));
21    }
22  }
```

initial node

create model

get shortest path

print node

number of flips

```
Enter an initial nine coins Hs and Ts: HHHTTTHHH ↵Enter

The steps to flip the coins are
HHH
TTT
HHH

HHH
THT
TTT

TTT
TTT
TTT

The number of flips is 8
```

The program prompts the user to enter an initial node with nine letters with a combination of `H`s and `T`s as a string in line 8, obtains an array of characters from the string (line 9), creates a model (line 11), obtains the shortest path from the initial node to the target node (lines 12–13), displays the nodes in the path (lines 16–17), and invokes `getNumberOfFlips` to get the number of flips needed to reach the target node (line 20).

Check Point

29.6.1 Why is the `tree` data field in `NineTailModel` in Listing 28.13 defined protected?

29.6.2 How are the nodes created for the graph in `WeightedNineTailModel`?

29.6.3 How are the edges created for the graph in `WeightedNineTailModel`?

KEY TERMS

Dijkstra's algorithm	1109
edge-weighted graph	1093
minimum spanning tree	1103
Prim's algorithm	1103
shortest path	1115
single-source shortest path	1110
vertex-weighted graph	1093

CHAPTER SUMMARY

1. You can use adjacency matrices or lists to store weighted edges in graphs.

2. A spanning tree of a graph is a subgraph that is a tree and connects all vertices in the graph.

3. Prim's algorithm for finding a minimum spanning tree works as follows: the algorithm starts with a spanning tree `T` that contains an arbitrary vertex. The algorithm expands the tree by adding a vertex with the minimum-weight edge incident to a vertex already in the tree.

4. Dijkstra's algorithm starts the search from the source vertex and keeps finding vertices that have the shortest path to the source until all vertices are found.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

***29.1** (*Kruskal's algorithm*) The text introduced Prim's algorithm for finding a minimum spanning tree. Kruskal's algorithm is another well-known algorithm for finding a minimum spanning tree. The algorithm repeatedly finds a minimum-weight edge and adds it to the tree if it does not cause a cycle. The process ends when all vertices are in the tree. Design and implement an algorithm for finding an MST using Kruskal's algorithm.

***29.2** (*Implement Prim's algorithm using an adjacency matrix*) The text implements Prim's algorithm using lists for adjacent edges. Implement the algorithm using an adjacency matrix for weighted graphs.

***29.3** (*Implement Dijkstra's algorithm using an adjacency matrix*) The text implements Dijkstra's algorithm using lists for adjacent edges. Implement the algorithm using an adjacency matrix for weighted graphs.

***29.4** (*Modify weight in the nine tails problem*) In the text, we assign the number of the flips as the weight for each move. Assuming the weight is three times of the number of flips, revise the program.

***29.5** (*Prove or disprove*) The conjecture is that both `NineTailModel` and `WeightedNineTailModel` result in the same shortest path. Write a program to prove or disprove it. (*Hint*: Let `tree1` and `tree2` denote the trees rooted at node `511` obtained from `NineTailModel` and `WeightedNineTailModel`, respectively. If the depth of a node `u` is the same in `tree1` and in `tree2`, the length of the path from `u` to the target is the same.)

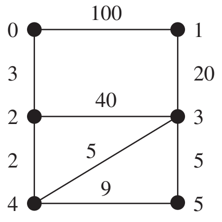
****29.6** (*Weighted 4 × 4 16 tails model*) The weighted nine tails problem in the text uses a 3 × 3 matrix. Assume that you have 16 coins placed in a 4 × 4 matrix. Create a new model class named `WeightedTailModel16`. Create an instance of the model and save the object into a file named WeightedTailModel16.dat.

****29.7** (*Weighted 4 × 4 16 tails*) Revise Listing 29.9, WeightedNineTail.java, for the weighted 4 × 4 16 tails problem. Your program should read the model object created from the preceding exercise.

****29.8** (*Traveling salesperson problem*) The traveling salesperson problem (TSP) is to find the shortest round-trip route that visits each city exactly once and then returns to the starting city. The problem is similar to finding a shortest Hamiltonian cycle in Programming Exercise 28.17. Add the following method in the `WeightedGraph` class:

```
// Return a shortest cycle
// Return null if no such cycle exists
public List<Integer> getShortestHamiltonianCycle()
```

***29.9** (*Find a minimum spanning tree*) Write a program that reads a connected graph from a file and displays its minimum spanning tree. The first line in the file contains a number that indicates the number of vertices (`n`). The vertices are labeled as `0`, `1`, . . . , `n-1`. Each subsequent line describes the edges in the form of `u1, v1, w1 | u2, v2, w2 | . . .` . Each triplet in this form describes an edge and its weight. Figure 29.23 shows an example of the file for the corresponding graph. Note we assume the graph is undirected. If the graph has an edge (`u`, `v`), it also has an edge (`v`, `u`). Only one edge is represented in the file. When you construct a graph, both edges need to be added.



(a)

```
File
6
0, 1, 100 | 0, 2, 3
1, 3, 20
2, 3, 40 | 2, 4, 2
3, 4, 5 | 3, 5, 5
4, 5, 9
```

(b)

FIGURE 29.23 The vertices and edges of a weighted graph can be stored in a file.

Your program should prompt the user to enter a URL for the file, read data from the file, create an instance `g` of `WeightedGraph`, invoke `g.printWeightedEdges()` to display all edges, invoke `getMinimumSpanningTree()` to obtain an instance `tree` of `WeightedGraph.MST`, invoke `tree.getTotalWeight()` to display the weight of the minimum spanning tree, and invoke `tree.printTree()` to display the tree. Here is a sample run of the program:

```
Enter a URL:
  https://liveexample.pearsoncmg.com/test/WeightedGraphSample.txt
The number of vertices is 6
Vertex 0: (0, 2, 3) (0, 1, 100)
Vertex 1: (1, 3, 20) (1, 0, 100)
Vertex 2: (2, 4, 2) (2, 3, 40) (2, 0, 3)
Vertex 3: (3, 4, 5) (3, 5, 5) (3, 1, 20) (3, 2, 40)
Vertex 4: (4, 2, 2) (4, 3, 5) (4, 5, 9)
Vertex 5: (5, 3, 5) (5, 4, 9)
Total weight in MST is 35
Root is: 0
Edges: (3, 1) (0, 2) (4, 3) (2, 4) (3, 5)
```

(*Hint*: Use `new WeightedGraph(list, numberOfVertices)` to create a graph, where `list` contains a list of `WeightedEdge` objects. Use `new WeightedEdge(u, v, w)` to create an edge. Read the first line to get the number of vertices. Read each subsequent line into a string `s` and use `s.split("[\\|]")` to extract the triplets. For each triplet, use `triplet.split("[,]")` to extract vertices and weight.)

***29.10** (*Create a file for a graph*) Modify Listing 29.3, TestWeightedGraph.java, to create a file for representing `graph1`. The file format is described in Programming Exercise 29.9. Create the file from the array defined in lines 7–24 in Listing 29.3. The number of vertices for the graph is `12`, which will be stored in the first line of the file. An edge (`u`, `v`) is stored if `u < v`. The contents of the file should be as follows:

```
12
0, 1, 807 | 0, 3, 1331 | 0, 5, 2097
1, 2, 381 | 1, 3, 1267
2, 3, 1015 | 2, 4, 1663 | 2, 10, 1435
3, 4, 599 | 3, 5, 1003
4, 5, 533 | 4, 7, 1260 | 4, 8, 864 | 4, 10, 496
5, 6, 983 | 5, 7, 787
6, 7, 214
7, 8, 888
8, 9, 661 | 8, 10, 781 | 8, 11, 810
9, 11, 1187
10, 11, 239
```

***29.11** (*Find shortest paths*) Write a program that reads a connected graph from a file. The graph is stored in a file using the same format specified in Programming Exercise 29.9. Your program should prompt the user to enter a URL for the file, then two vertices, and should display a shortest path between the two vertices. For example, for the graph in Figure 29.23, a shortest path between `0` and `1` can be displayed as `0 2 4 3 1`. Here is a sample run of the program:

```
Enter a URL:
  https://liveexample.pearsoncmg.com/test/WeightedGraphSample2.txt ↵ Enter
Enter two vertices (integer indexes): 0 1 ↵ Enter
The number of vertices is 6
Vertex 0: (0, 2, 3) (0, 1, 100)
Vertex 1: (1, 3, 20) (1, 0, 100)
Vertex 2: (2, 4, 2) (2, 3, 40) (2, 0, 3)
Vertex 3: (3, 4, 5) (3, 5, 5) (3, 1, 20) (3, 2, 40)
Vertex 4: (4, 2, 2) (4, 3, 5) (4, 5, 9)
Vertex 5: (5, 3, 5) (5, 4, 9)
A path from 0 to 1: 0 2 4 3 1
```

***29.12** (*Display weighted graphs*) Revise **`GraphView`** in Listing 28.6 to display a weighted graph. Write a program that displays the graph in Figure 29.1 as shown in Figure 29.24. (Instructors may ask students to expand this program by adding new cities with appropriate edges into the graph.)

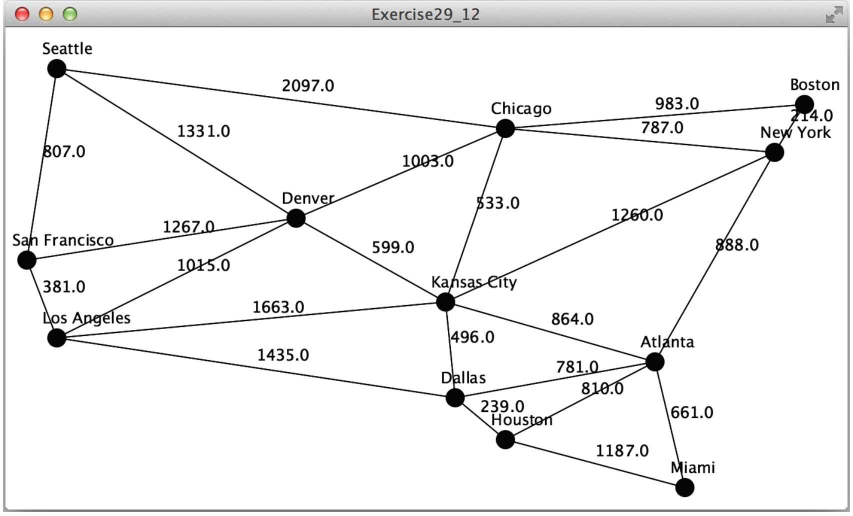
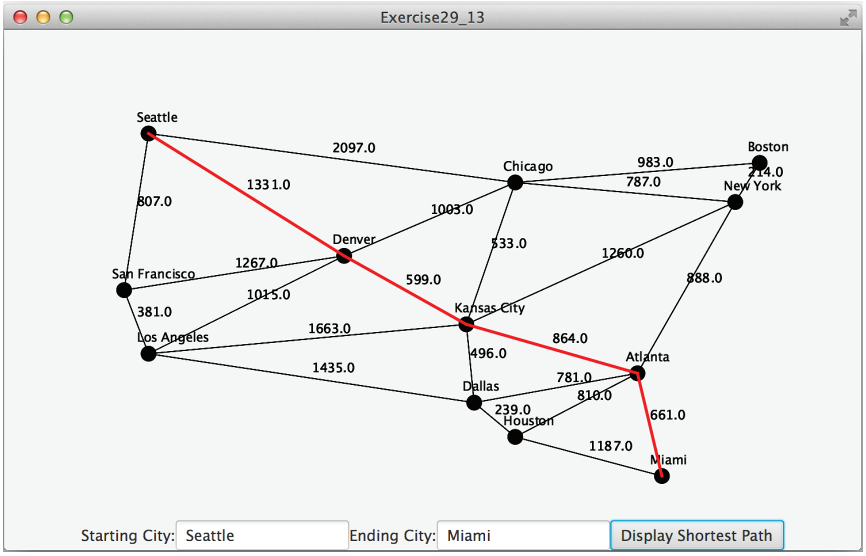

Exercise29_12
Seattle
2097.0
Chicago
983.0
Boston
214.0
1331.0
787.0
New York
807.0
1003.0
Denver
533.0
1260.0
1267.0
San Francisco
599.0
888.0
1015.0
Kansas City
381.0
1663.0
864.0
Los Angeles
496.0
Atlanta
1435.0
781.0
Dallas
810.0
239.0
Houston
661.0
1187.0
Miami

FIGURE 29.24 Programming Exercise 29.12 displays a weighted graph.

***29.13** (*Display shortest paths*) Revise **`GraphView`** in Listing 28.6 to display a weighted graph and a shortest path between the two specified cities, as shown in Figure 29.25. You need to add a data field **`path`** in **`GraphView`**. If a **`path`** is not null, the edges in the path are displayed in red. If a city not in the map is entered, the program displays a text to alert the user.


Exercise29_13
Seattle
2097.0
Chicago
983.0
Boston
214.0
1331.0
787.0
New York
807.0
1003.0
Denver
533.0
1260.0
1267.0
San Francisco
599.0
888.0
1015.0
Kansas City
381.0
1663.0
864.0
Los Angeles
496.0
Atlanta
1435.0
781.0
Dallas
810.0
239.0
Houston
661.0
1187.0
Miami
Display Shortest Path

Starting City: Seattle
Ending City: Miami

FIGURE 29.25 Programming Exercise 29.13 displays a shortest path.

***29.14** (*Display a minimum spanning tree*) Revise `GraphView` in Listing 28.6 to display a weighted graph and a minimum spanning tree for the graph in Figure 29.1, as shown in Figure 29.26. The edges in the MST are shown in red.

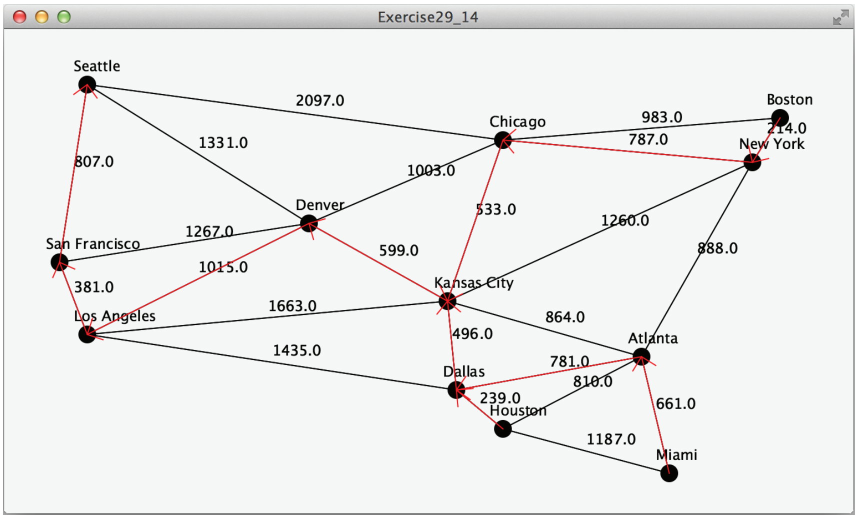
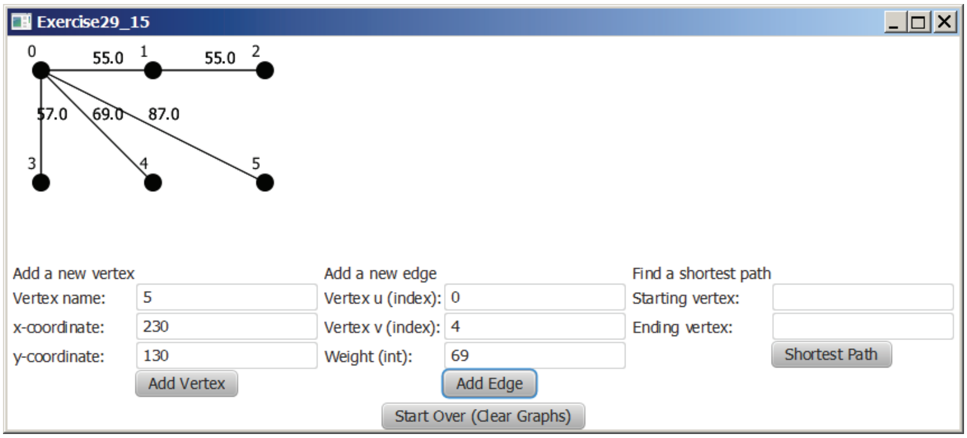

Exercise29_14
Seattle
2097.0
Chicago
983.0
Boston
1331.0
214.0
787.0
New York
807.0
1003.0
Denver
533.0
1260.0
San Francisco
1267.0
599.0
888.0
1015.0
381.0
Kansas City
Los Angeles
1663.0
864.0
496.0
Atlanta
1435.0
781.0
Dallas
810.0
239.0
Houston
661.0
1187.0
Miami

FIGURE 29.26 Programming Exercise 29.14 displays an MST.

*****29.15** (*Dynamic graphs*) Write a program that lets the users create a weighted graph dynamically. The user can create a vertex by entering its name and location, as shown in Figure 29.27. The user can also create an edge to connect two vertices. To simplify the program, assume vertex names are the same as vertex indices. You have to add the vertex indices **0**, **1**, . . . , and **n**, in this order. The user can specify two vertices and let the program display their shortest path in red.


Exercise29_15
0
55.0
1
55.0
2
57.0
69.0
87.0
3
4
5
Add Vertex
Add Edge
Shortest Path
Start Over (Clear Graphs)

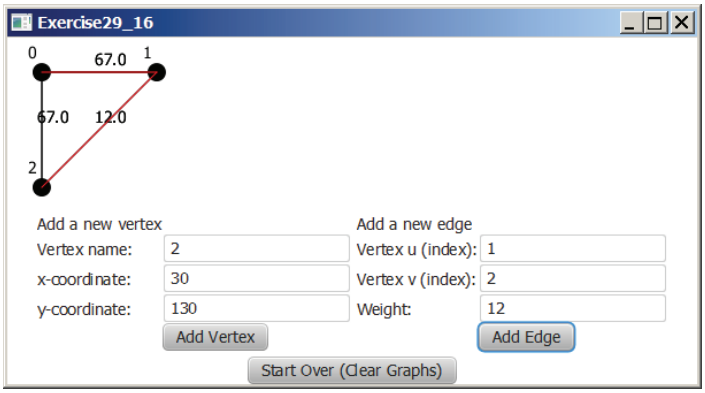
Add a new vertex
Vertex name: 5
x-coordinate: 230
y-coordinate: 130

Add a new edge
Vertex u (index): 0
Vertex v (index): 4
Weight (int): 69

Find a shortest path
Starting vertex:
Ending vertex:

FIGURE 29.27 The program can add vertices and edges and display a shortest path between two specified vertices. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

*****29.16** (*Display a dynamic MST*) Write a program that lets the user create a weighted graph dynamically. The user can create a vertex by entering its name and location, as shown in Figure 29.28. The user can also create an edge to connect two vertices. To simplify the program, assume vertex names are the same as those of vertex indices. You have to add the vertex indices **0**, **1**, . . . , and **n**, in this order. The edges in the MST are displayed in red. As new edges are added, the MST is redisplayed.



Add a new vertex
Vertex name: 2
x-coordinate: 30
y-coordinate: 130

Add a new edge
Vertex u (index): 1
Vertex v (index): 2
Weight: 12

FIGURE 29.28 The program can add vertices and edges and display MST dynamically. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

*****29.17** (*Weighted graph visualization tool*) Develop a GUI program as shown in Figure 29.2, with the following requirements: (1) The radius of each vertex is 20 pixels. (2) The user clicks the left mouse button to place a vertex centered at the mouse point, provided the mouse point is not inside or too close to an existing vertex. (3) The user clicks the right mouse button inside an existing vertex to remove the vertex. (4) The user presses a mouse button inside a vertex and drags to another vertex then releases the button to create an edge, and the distance between the two vertices is also displayed. (5) The user drags a vertex while pressing the *CTRL* key to move a vertex. (6) The vertices are numbers starting from **0**. When a vertex is removed, the vertices are renumbered. (7) You can click the *Show MST* or *Show All SP From the Source* button to display an MST or SP tree from a starting vertex. (8) You can click the *Show Shortest Path* button to display the shortest path between the two specified vertices.

*****29.18** (*Alternative version of Dijkstra algorithm*) An alternative version of the Dijkstra algorithm can be described as follows:

```
Input: a weighted graph G = (V, E) with nonnegative weights
Output: A shortest-path tree from a source vertex s

ShortestPathTree getShortestPath(s) {
  Let T be a set that contains the vertices whose
    paths to s are known;
  Initially T contains source vertex s with cost[s] = 0;
  for (each u in V – T)
     cost[u] = infinity;

  while (size of T < n) {
    Find v in V – T with the smallest cost[u] + w(u, v) value
```

add initial vertex

more vertex

find next vertex

add a vertex

```
          among all u in T;
        Add v to T and set cost[v] = cost[u] + w(u, v);
        parent[v] = u;
      }
    }
```

The algorithm uses `cost[v]` to store the cost of a shortest path from vertex `v` to the source vertex `s`. `cost[s]` is `0`. Initially assign infinity to `cost[v]` to indicate that no path is found from `v` to `s`. Let `V` denote all vertices in the graph and `T` denote the set of the vertices whose costs are known. Initially, the source vertex `s` is in `T`. The algorithm repeatedly finds a vertex `u` in `T` and a vertex `v` in `V–T` such that `cost[u] + w(u, v)` is the smallest, and moves `v` to `T`. The shortest-path algorithm given in the text continuously updates the cost and parent for a vertex in `V–T`. This algorithm initializes the cost to infinity for each vertex and then changes the cost for a vertex only once when the vertex is added into `T`. Implement this algorithm and use Listing 29.7, TestShortestPath.java, to test your new algorithm.

*****29.19** (*Find* `u` *with smallest* `cost[u]` *efficiently*) The `getShortestPath` method finds a `u` with the smallest `cost[u]` using a linear search, which takes $O(|V|)$. The search time can be reduced to $O(\log|V|)$ using an AVL tree. Modify the method using an AVL tree to store the vertices in `V–T`. Use Listing 29.7, TestShortestPath.java, to test your new implementation.

*****29.20** (*Test if a vertex* `u` *is in T efficiently*) Since `T` is implemented using a list in the `getMinimumSpanningTree` and `getShortestPath` methods in Listing 29.2, WeightedGraph.java, testing whether a vertex `u` is in `T` by invoking `T.contains(u)` takes `O(n)` time. Modify these two methods by introducing an array named `isInT`. Set `isInT[u]` to `true` when a vertex `u` is added to `T`. Testing whether a vertex `u` is in `T` can now be done in `O(1)` time. Write a test program using the following code, where `graph1` is created from Figure 29.1:

```
WeightedGraph<String> graph1 = new WeightedGraph<>(edges, vertices);

WeightedGraph<String>.MST tree1 = graph1.getMinimumSpanningTree();

System.out.println("Total weight is " + tree1.getTotalWeight());

tree1.printTree();

WeightedGraph<String>.ShortestPathTree tree2 =
  graph1.getShortestPath(graph1.getIndex("Chicago"));

tree2.printAllPaths();
```

CHAPTER

30

AGGREGATE OPERATIONS FOR COLLECTION STREAMS

Objectives

- To use aggregate operations on collection streams to simplify coding and improve performance (§30.1).
- To create a stream pipeline, apply lazy intermediate methods (**`skip`**, **`limit`**, **`filter`**, **`distinct`**, **`sorted`**, **`map`**, and **`mapToInt`**), and terminal methods (**`count`**, **`sum`**, **`average`**, **`max`**, **`min`**, **`forEach`**, **`findFirst`**, **`firstAny`**, **`anyMatch`**, **`allMatch`**, **`noneMatch`**, and **`toArray`**) on a stream (§30.2).
- To process primitive data values using the **`IntStream`**, **`LongStream`**, and **`DoubleStream`** (§30.3).
- To create parallel streams for fast execution (§30.4).
- To reduce the elements in a stream into a single result using the **`reduce`** method (§30.5).
- To place the elements in a stream into a mutable collection using the **`collect`** method (§30.6).
- To group the elements in a stream and apply aggregate methods for the elements in the groups (§30.7).
- To use a variety of examples to demonstrate how to simplify coding using streams (§30.8).

30.1 Introduction

Key Point

Using aggregate operations on collection streams can greatly simplify coding and improve performance.

Often, you need to process data in an array or a collection. Suppose, for instance, that you need to count the number of elements in a set that is greater than `60`. You may write the code using a foreach loop as follows:

```
Double[] numbers = {2.4, 55.6, 90.12, 26.6};
Set<Double> set = new HashSet<>(Arrays.asList(numbers));
int count = 0;
for (double e: set)
  if (e > 60)
    count++;
System.out.println("Count is " + count);
```

The code is fine. However, Java provides a better and simpler way for accomplishing the task. Using the aggregate operations, you can rewrite the code as follows:

```
System.out.println("Count is "
  + set.stream().filter(e -> e > 60).count());
```

Invoking the `stream()` method on a set returns a `Stream` for the elements from a set. The `filter` method specifies a condition for selecting the elements whose value is greater than `60`. The `count()` method returns the number of elements in the stream that satisfy the condition.

stream

aggregate operations

A *collection stream* or simply *stream* is a sequence of elements. The operations on a stream is called *aggregate operations* (also known as *stream operations*) because they apply to all the data in the stream. The `filter` and `count` are the examples of aggregate operations. The code written using a foreach loop describes the process how to obtain the count, that is, for each element, if it is greater than `60`, increase the count. The code written using the aggregate operations tells the program to return the count for the elements greater than `60`, but it does not specify how the count is obtained. Clearly, using the aggregate operations leaves the detailed implementation to the computer, therefore, makes the code concise and simpler. Moreover, the aggregate operations on a stream can be executed in parallel to take advantage of multiple processors. So, the code written using aggregate operations usually run faster than the ones using a foreach loop.

why using aggregate operations?

Java provides many aggregate operations and many different ways of using aggregate operations. This chapter gives a comprehensive coverage on aggregate operations and streams.

Check Point

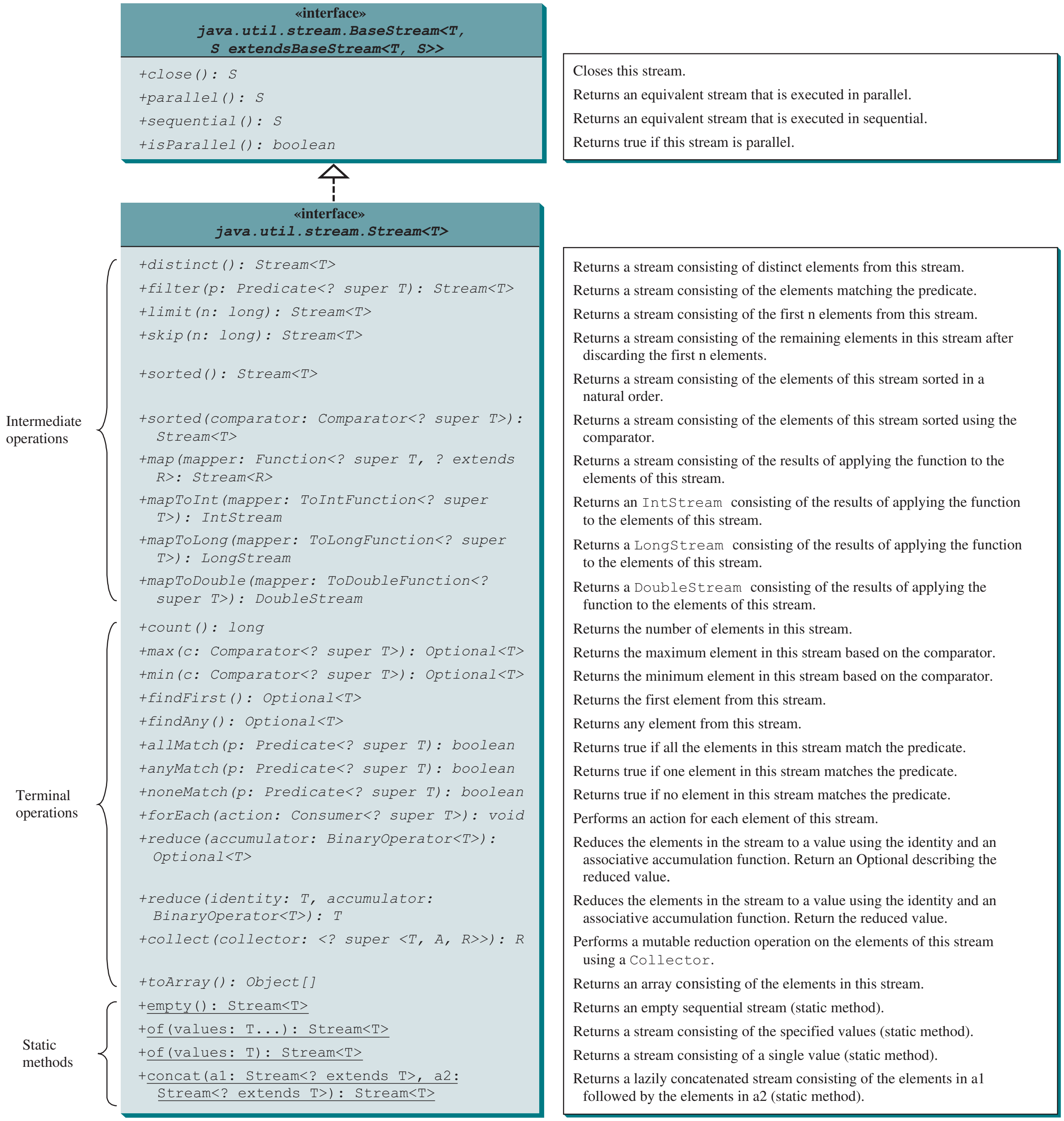
30.1.1 What are the benefits of using aggregate operations on collection streams for processing data?

30.2 Stream Pipelines

Key Point

A stream pipeline consists of a stream created from a data source, zero or more intermediate methods, and a final terminal method.

An array or a collection is an object for storing data. A stream is a transient object for processing data. After data is processed, the stream is destroyed. Java 8 introduced a new default `stream()` method in the `Collection` interface to return a `Stream` object. The `Stream` interface extends the `BaseStream` interface and contains the aggregate methods and the utility methods as shown in Figure 30.1.


«interface»
java.util.stream.BaseStream<T, S extendsBaseStream<T, S>>
«interface»
java.util.stream.Stream<T>
Intermediate operations
Terminal operations
Static methods

`+close(): S`	Closes this stream.
`+parallel(): S`	Returns an equivalent stream that is executed in parallel.
`+sequential(): S`	Returns an equivalent stream that is executed in sequential.
`+isParallel(): boolean`	Returns true if this stream is parallel.

`+distinct(): Stream<T>`	Returns a stream consisting of distinct elements from this stream.
`+filter(p: Predicate<? super T): Stream<T>`	Returns a stream consisting of the elements matching the predicate.
`+limit(n: long): Stream<T>`	Returns a stream consisting of the first n elements from this stream.
`+skip(n: long): Stream<T>`	Returns a stream consisting of the remaining elements in this stream after discarding the first n elements.
`+sorted(): Stream<T>`	Returns a stream consisting of the elements of this stream sorted in a natural order.
`+sorted(comparator: Comparator<? super T>): Stream<T>`	Returns a stream consisting of the elements of this stream sorted using the comparator.
`+map(mapper: Function<? super T, ? extends R>: Stream<R>`	Returns a stream consisting of the results of applying the function to the elements of this stream.
`+mapToInt(mapper: ToIntFunction<? super T>): IntStream`	Returns an `IntStream` consisting of the results of applying the function to the elements of this stream.
`+mapToLong(mapper: ToLongFunction<? super T>): LongStream`	Returns a `LongStream` consisting of the results of applying the function to the elements of this stream.
`+mapToDouble(mapper: ToDoubleFunction<? super T>): DoubleStream`	Returns a `DoubleStream` consisting of the results of applying the function to the elements of this stream.
`+count(): long`	Returns the number of elements in this stream.
`+max(c: Comparator<? super T>): Optional<T>`	Returns the maximum element in this stream based on the comparator.
`+min(c: Comparator<? super T>): Optional<T>`	Returns the minimum element in this stream based on the comparator.
`+findFirst(): Optional<T>`	Returns the first element from this stream.
`+findAny(): Optional<T>`	Returns any element from this stream.
`+allMatch(p: Predicate<? super T): boolean`	Returns true if all the elements in this stream match the predicate.
`+anyMatch(p: Predicate<? super T): boolean`	Returns true if one element in this stream matches the predicate.
`+noneMatch(p: Predicate<? super T): boolean`	Returns true if no element in this stream matches the predicate.
`+forEach(action: Consumer<? super T>): void`	Performs an action for each element of this stream.
`+reduce(accumulator: BinaryOperator<T>): Optional<T>`	Reduces the elements in the stream to a value using the identity and an associative accumulation function. Return an Optional describing the reduced value.
`+reduce(identity: T, accumulator: BinaryOperator<T>): T`	Reduces the elements in the stream to a value using the identity and an associative accumulation function. Return the reduced value.
`+collect(collector: <? super <T, A, R>>): R`	Performs a mutable reduction operation on the elements of this stream using a `Collector`.
`+toArray(): Object[]`	Returns an array consisting of the elements in this stream.
`+empty(): Stream<T>`	Returns an empty sequential stream (static method).
`+of(values: T...): Stream<T>`	Returns a stream consisting of the specified values (static method).
`+of(values: T): Stream<T>`	Returns a stream consisting of a single value (static method).
`+concat(a1: Stream<? extends T>, a2: Stream<? extends T>): Stream<T>`	Returns a lazily concatenated stream consisting of the elements in a1 followed by the elements in a2 (static method).

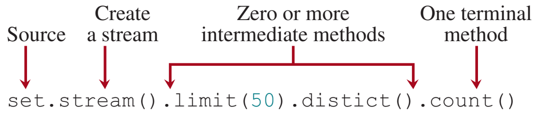
FIGURE 30.1 The **`Stream`** interface defines the aggregate operations for the elements in a stream.

The methods in the **`Stream`** interface are divided into three groups: intermediate methods, terminal methods, and static methods. An *intermediate method* transforms the stream into another stream. A *terminal method* returns a result or performs actions. After a terminal method is executed, the stream is closed automatically. A *static method* creates a stream.

intermediate method
terminal method
static method

stream pipeline

The methods are invoked using a stream pipeline. A *stream pipeline* consists of a source (e.g., a list, a set, or an array), a method that creates a stream, zero or more intermediate methods, and a final terminal method. The following is a stream pipeline example:



```
set.stream().limit(50).distict().count()
```

Here, **`set`** is the source of the data, invoking **`stream()`** creates a stream for the data from the source, invoking **`limit(50)`** returns the first **`50`** elements from the stream, invoking **`distinct()`** obtains a stream of distinct elements from the stream of the **`50`** elements, and invoking **`count()`** returns the number of elements in the final stream.

lazy evaluation

Streams are lazy, which means that the computation is performed only when the terminal operation is initiated. This allows the JVM to optimize computation.

Most of the arguments for stream methods are instances of functional interfaces. So the arguments can be created using lambda expressions or method references. Listing 30.1 gives an example that demonstrates creating a stream and applying methods on the streams.

LISTING 30.1 `StreamDemo.java`

```
import java.util.stream.Stream;

public class StreamDemo {
  public static void main(String[] args) {
    String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
      "George", "Alan", "Stacy", "Michelle", "john"};

    // Display the first four names sorted
    Stream.of(names).limit(4).sorted()
      .forEach(e -> System.out.print(e + " "));

    // Skip four names and display the rest sorted ignore case
    System.out.println();
    Stream.of(names).skip(4)
      .sorted((e1, e2) -> e1.compareToIgnoreCase(e2))
      .forEach(e -> System.out.print(e + " "));

    System.out.println();
    Stream.of(names).skip(4)
      .sorted(String::compareToIgnoreCase)
      .forEach(e -> System.out.print(e + " "));

    System.out.println("\nLargest string with length > 4: "
      + Stream.of(names)
        .filter(e -> e.length() > 4)
        .max(String::compareTo).get());

    System.out.println("Smallest string alphabetically: "
      + Stream.of(names).min(String::compareTo).get());

    System.out.println("Stacy is in names? "
      + Stream.of(names).anyMatch(e -> e.equals("Stacy")));

    System.out.println("All names start with a capital letter? "
      + Stream.of(names)
        .allMatch(e -> Character.isUpperCase(e.charAt(0))));

```

create an array
`Stream.of`
`forEach`
`skip`
`sorted`
method reference
`filter`
method reference
`min`
`anyMatch`
`allMatch`

```
38      System.out.println("No name begins with Ko? "
39        + Stream.of(names).noneMatch(e -> e.startsWith("Ko")));
40
41      System.out.println("Number of distinct case-insensitive strings: "
42        + Stream.of(names).map(e -> e.toUpperCase())
43          .distinct().count());
44
45      System.out.println("First element in this stream in lowercase: "
46        + Stream.of(names).map(String::toLowerCase).findFirst().get());
47
48      System.out.println("Skip 4 and get any element in this stream:"
49        + Stream.of(names).skip(4).sorted().findAny().get());
50
51      Object[] namesInLowerCase =
52        Stream.of(names).map(String::toLowerCase).toArray();
53      System.out.println(java.util.Arrays.toString(namesInLowerCase));
54    }
55  }
```

`distinct`

`findFirst`

`findAny`

`toArray`

display array

```
John Kim Peter Susan
Alan George Jen john Michelle Stacy
Alan George Jen john Michelle Stacy
Largest string with length > 4: Susan
Smallest string alphabetically: Alan
Stacy is in names? true
All names start with a capital letter? false
No name begins with Ko? true
Number of distinct case-insensitive strings: 9
First element in this stream in lowercase: john
Skip 4 and get any element in this stream: Alan
[john, peter, susan, kim, jen, george, alan, stacy, michelle, john]
```

30.2.1 The **`Stream.of`**, **`limit`**, and **`forEach`** Methods

The program creates an array of strings (lines 5–6). In lines 9–10, invoking the static **`Stream.of(names)`** returns a **`Stream`** consisting of strings from the **`names`** array, invoking **`limit(4)`** returns a new **`Stream`** consisting the first four elements in the stream, invoking **`sorted()`** sorts the stream, and invoking the **`forEach`** method displays each element in the stream. The argument passed to the **`forEach`** method is a lambda expression. As introduced in Section 15.6, a lambda expression is a concise syntax to replace an anonymous inner class that implements a functional interface. The argument passed to the **`forEach`** method is an instance of the functional interface **`Consumer<? super T>`** with an abstract function **`accept(T t)`**. The statement in line 10 using a lambda expression in (a) is equivalent to using an anonymous inner class in (b) as shown below. The lambda expression not only simplifies the code, but also the concept of the method. You can now simply say that for each element in the stream, perform the action as specified in the expression.

`forEach` method

lambda expression

```
forEach(e -> System.out.print(e + " "))
```

(a) Using a lambda expression

```
forEach(
  new java.util.function.Consumer<String>() {
    public void accept(String e) {
      System.out.print(e + " ");
    }
  }
)
```

(b) Using an anonymous inner class

30.2.2 The sorted Method

The `sorted` method in line 15 sorts the strings in the stream using a `Comparator`. The `Comparator` is a functional interface. A lambda expression is used to implement the interface and specifies that two strings are compared ignoring cases. This lambda expression in (a) is equivalent to the code using an anonymous inner class in (b). The lambda expression simply invokes a method in this case. So, it can be further simplified using a method reference in line 20 (also see in (c)). The method reference was introduced in Section 20.6.

method reference

```
sorted((e1, e2) ->
  e1.compareToIgnoreCase(e2))
```

(a) Using a lambda expression

```
sorted(String::compareToIgnoreCase)
```

(c) Using a method reference

```
sorted(
  new java.util.Comparator<String>() {
    public int compare(String e1, String e2) {
      return e1.compareToIgnoreCase(e2);
    }
  }
)
```

(b) Using an anonymous inner class

30.2.3 The `filter` Method

The `filter` method takes an argument of the `Predicate<? super T>` type, which is a functional interface with an abstract method `test(T t)` that returns a Boolean value. The method selects the elements from the stream that satisfies the predicate. Line 25 uses a lambda expression to implement the `Predicate` interface as shown in (a), which is equivalent to the code using an anonymous inner class as shown in (b).

```
filter(e -> e.length() > 4)
```

(a) Using a lambda expression

```
filter(
  new java.util.function.Predicate<String>() {
    public boolean test(String e) {
      return e.length() > 4;
    }
  }
)
```

(b) Using an anonymous inner class

30.2.4 The `max` and `min` Methods

The `max` and `min` methods take an argument of the `Comparator<? Super T>` type. This argument specifies how the elements are compared in order to obtain the maximum and minimum elements. The program uses the method reference `String::compareTo` to simplify the code for creating a `Comparator` (lines 26 and 29). The `max` and `min` methods return an `Optional<T>` that describes the element. You need to invoke the `get()` method from the `Optional` class to return the element.

`Optional <T>`

`get` method

30.2.5 The `anyMatch`, `allMatch`, and `noneMatch` Methods

The `anyMatch`, `allMatch`, and `noneMatch` methods take an argument of the `Predicate<? super T>` type to test if the stream contains an element, all elements, or no element that satisfies the predicate. The program tests whether the name `Stacy` is in the stream (line 32), whether all the names in the stream start with a capital letter (line 36), and whether any names starts with string `Ko` (line 39).

30.2.6 The `map`, `distinct`, and `count` Methods

The `map` method returns a new stream by mapping each element in the stream into a new element. So, the `map` method in line 42 returns a new stream with all uppercase strings. The `distinct()` method obtains a new stream with all distinct elements. The `count()` method counts the number of the elements in the stream. So, the stream pipeline in line 43 counts the number of distinct strings in the array `names`.

The `map` method takes an argument of the `Function<? super T, ? super R>` type to return an instance of the `Stream<R>`. The `Function` is a functional interface with an abstract method `apply(T t)` that maps `t` into a value of the type `R`. Line 42 uses a lambda expression to implement the `Function` interface as shown in (a), which is equivalent to the code using an anonymous inner class as shown in (b). You can further simplify it using a method reference as shown in (c).

```
map(e -> e.toUpperCase())
```

(a) Using a lambda expression

```
map(
  new java.util.function.Function<String, String>() {
    public String apply(String e) {
      return e.toUpperCase();
    }
  }
)
```

(b) Using an anonymous inner class

```
map(String::toUpperCase)
```

(c) Using a method reference

30.2.7 The `findFirst`, `findAny`, and `toArray` Methods

The `findFirst()` method (line 46) returns the first element in the stream wrapped in an instance of `Optional<T>`. The actual element value is then returned by invoking the `get()` method in the `Optional<T>` class. The `findAny()` method (line 49) returns any element in the stream. Which element is selected depends on the internal state of the stream. The `findAny()` method is more efficient than the `findFirst()` method.

The `toArray()` method (line 52) returns an array of objects from the stream.

Note

The `BaseStream` interface defines the `close()` method, which can be invoked to close a stream. You don't need to use it because the stream is automatically closed after the terminal method is executed.

`close` method

Check Point

30.2.1 Show the output of the following code:

```
Character[] chars = {'D', 'B', 'A', 'C'};
System.out.println(Stream.of(chars).sorted().findFirst().get());
System.out.println(Stream.of(chars).sorted(
  java.util.Comparator.reverseOrder()).findFirst().get());
System.out.println(Stream.of(chars)
  .limit(2).sorted().findFirst().get());
System.out.println(Stream.of(chars).distinct()
  .skip(2).filter(e -> e > 'A').findFirst().get());
System.out.println(Stream.of(chars)
  .max(Character::compareTo).get());
System.out.println(Stream.of(chars)
  .max(java.util.Comparator.reverseOrder()).get());
System.out.println(Stream.of(chars)
  .filter(e -> e > 'A').findFirst().get());
System.out.println(Stream.of(chars)
  .allMatch(e -> e >= 'A'));
System.out.println(Stream.of(chars)
  .anyMatch(e -> e > 'F'));
System.out.println(Stream.of(chars)
  .noneMatch(e -> e > 'F'));
Stream.of(chars).map(e -> e + "").map(e -> e.toLowerCase())
  .forEach(System.out::println);
```

```
Object[] temp = Stream.of(chars).map(e -> e + "Y")
  .map(e -> e.toLowerCase()).sorted().toArray();
System.out.println(java.util.Arrays.toString(temp));
```

30.2.2 What is wrong in the following code?

```
Character[] chars = {'D', 'B', 'A', 'C'};
Stream<Character> stream = Stream.of(chars).sorted();
System.out.println(stream.findFirst());
System.out.println(stream.skip(2).findFirst());
```

30.2.3 Rewrite (a) using a method reference and an anonymous inner class and (b) using lambda expression and an anonymous inner class:

```
(a) sorted((s1, s2) -> s1.compareToIgnoreCase(s2))
(b) forEach(System.out::println)
```

30.2.4 Given a `map` of the type `Map<String, Double>`, write an expression that returns the sum of all the values in `map`. For example, if the `map` contains `{"john", 1.5}` and `{"Peter", 1.1}`, the sum is `2.6`.

30.3 `IntStream`, `LongStream`, and `DoubleStream`

Key Point

`IntStream`, `LongStream`, and `DoubleStream` are special type of streams for processing a sequence of primitive `int`, `long`, and `double` values.

`IntStream`
`LongStream`
`DoubleStream`

`Stream` represents a sequence of objects. In addition to `Stream`, Java provides `IntStream`, `LongStream`, and `DoubleStream` for representing a sequence of `int`, `long`, and `double` values. These streams are also subinterfaces of `BaseStream`. You can use these streams in the same way like a `Stream`. Additionally, you can use the `sum()`, `average()`, and `summaryStatistics()` methods for returning the sum, average, and various statistics of the elements in the stream. You can use the `mapToInt` method to convert a `Stream` to an `IntStream` and use the `map` method to convert any stream including an `IntStream` to a `Stream`.

Listing 30.2 gives an example of using the `IntStream`.

Listing 30.2 `IntStreamDemo.java`

create an `int` array
`IntStream.of`
`average`
`sum`
`summaryStatistics`

```
import java.util.IntSummaryStatistics;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class IntStreamDemo {
  public static void main(String[] args) {
    int[] values = {3, 4, 1, 5, 20, 1, 3, 3, 4, 6};

    System.out.println("The average of distinct even numbers > 3: " +
      IntStream.of(values).distinct()
        .filter(e -> e > 3 && e %  2 == 0).average().getAsDouble());

    System.out.println("The sum of the first 4 numbers is " +
      IntStream.of(values).limit(4).sum());

    IntSummaryStatistics stats =
    IntStream.of(values).summaryStatistics();

    System.out.printf("The summary of the stream is\n%-10s%10d\n" +
      "%-10s%10d\n%-10s%10d\n%-10s%10d\n%-10s%10.2f\n",
21       " Count:", stats.getCount(), " Max:", stats.getMax(),
22       " Min:", stats.getMin(), " Sum:", stats.getSum(),
23       " Average:", stats.getAverage());
24
25     String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
26       "George", "Alan", "Stacy", "Michelle", "john"};
27
28     System.out.println("Total character count for all names is "
29       + Stream.of(names).mapToInt(e -> e.length()).sum());
30
31     System.out.println("The number of digits in array values is " +
32       Stream.of(values).map(e -> e + "")
33         .mapToInt(e -> e.length()).sum());
34   }
35 }
```

mapToInt

map

mapToInt

```
The average of distinct even numbers > 3: 10.0
The sum of the first 4 numbers is 13
The summary of the stream is

  Count:          10
  Max:            20
  Min:             1
  Sum:            50
  Average:      5.00

Total character count for all names is 47
The number of digits in array values is 11
```

The program creates an array of **int** values (line 7). The stream pipeline in line 10 applies the intermediate methods **distinct** and **filter** with a terminal method **average**. The **average()** method returns the average value in the stream as an **OptionalDouble** object (line 11). The actual average value is obtained by invoking the **getAsDouble()** method.

OptionalDouble class

average method

The stream pipeline in line 14 applies the intermediate method **limit** with a terminal method **sum**. The **sum()** method returns the sum of all values in the stream.

sum method

If you need to obtain multiple summary values from the stream, using the **summaryStatistics()** method is more efficient. This method (line 17) returns an instance of **IntSummaryStatistics** that contains summary values for count, min, max, sum, and average (lines 19–23). Note **sum()**, **average()**, and **summaryStatistics()** methods are only applicable to the **IntStream**, **LongStream**, and **DoubleStream**.

summaryStatistics method

The **mapToInt** method returns an **IntStream** by mapping each element in the stream into an **int** value. The **mapToInt** method in the stream pipeline in line 29 maps each string into an **int** value that is the length of a string, and the **sum** method obtains the sum of all the **int** values in the **IntStream**. The stream pipeline in line 29 obtains the total count for all characters in the stream.

The **mapToInt** method takes an argument of the **ToIntFunction<? super T>** type to return an instance of the **IntStream**. The **ToIntFunction** is a functional interface with an abstract method **applyAsInt(T t)** that maps **t** into a value of the type **int**. Line 33 uses a lambda expression to implement the **ToIntFunction** interface as shown in (a), which is equivalent to the code using an anonymous inner class as shown in (b). You can also further simplify it using a method reference as shown in (c).

mapToInt method

```
mapToInt(e -> e.length())
```

(a) Using a lambda expression

```
mapToInt(String::length)
```

(c) Using a method reference

```
mapToInt(
  new java.util.function.ToIntFunction<String>() {
    public int applyAsInt(String e) {
      return e.length();
    }
  }
)
```

(b) Using an anonymous inner class

The `map` method in line 32 returns a new stream of strings. Each string is converted from an integer in array `values`. The `mapToInt` method in line 33 returns a new stream of integers. Each integer represents the length of a string. The `sum()` method returns the sum of all `int` values in the final stream. So the stream pipeline in lines 32–33 obtains the total number of digits in the array `values`.

Check Point

30.3.1 Show the output of the following code:

```
int[] numbers = {1, 4, 2, 3, 1};
System.out.println(IntStream.of(numbers)
  .sorted().findFirst().getAsInt());
System.out.println(IntStream.of(numbers)
  .limit(2).sorted().findFirst().getAsInt());
System.out.println(IntStream.of(numbers).distinct()
  .skip(1).filter(e -> e > 2).sum());
System.out.println(IntStream.of(numbers).distinct()
  .skip(1).filter(e -> e > 2).average().getAsDouble());
System.out.println(IntStream.of(numbers).max().getAsInt());
System.out.println(IntStream.of(numbers).max().getAsInt());
System.out.println(IntStream.of(numbers)
  .filter(e -> e > 1).findFirst().getAsInt());
System.out.println(IntStream.of(numbers)
  .allMatch(e -> e >= 1));
System.out.println(IntStream.of(numbers)
  .anyMatch(e -> e > 4));
System.out.println(IntStream.of(numbers).noneMatch(e -> e > 4));
IntStream.of(numbers).mapToObj(e -> (char)(e + 50))
  .forEach(System.out::println);

Object[] temp = IntStream.of(numbers)
  .mapToObj(e -> (char)(e + 'A')).toArray();
System.out.println(java.util.Arrays.toString(temp));
```

30.3.2 What is wrong in the following code?

```
int[] numbers = {1, 4, 2, 3, 1};
DoubleSummaryStatistics stats =
  DoubleStream.of(numbers).summaryStatistics();
System.out.printf("The summary of the stream is\n%-10s%10d\n" +
  "%-10s%10.2f\n%-10s%10.2f\n%-10s%10.2f\n%-10s%10.2f\n",
  " Count:", stats.getCount(), " Max:", stats.getMax(),
  " Min:", stats.getMin(), " Sum:", stats.getSum(),
  " Average:", stats.getAverage());
```

30.3.3 Rewrite the following code that maps an `int` to a `Character` using an anonymous inner class:

```
mapToObj(e -> (char)(e + 50))
```

30.3.4 Show the output of the following code:

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}};
System.out.println(Stream.of(m)
  .mapToInt(e -> IntStream.of(e).sum()).sum());
```

30.3.5 Given an array `names` in Listing 30.1, write the code to display the total number of characters in `names`.

30.4 Parallel Streams

Streams can be executed in parallel mode to improve performance.

Key Point

The widespread use of multicore systems has created a revolution in software. In order to benefit from multiple processors, software needs to run in parallel. All stream operations can execute in parallel to utilize the multicore processors. The `stream()` method in the `Collection` interface returns a sequential stream. To execute operations in parallel, use the `parallelStream()` method in the `Collection` interface to obtain a parallel stream. Any stream can be turned into a parallel stream by invoking the `parallel()` method defined in tbe `BaseStream` interface. Likewise, you can turn a parallel stream into a sequential stream by invoking the `sequential()` method.

Intermediate methods can be further divided into *stateless* and *stateful* methods. A stateless method such as `filter` and `map` can be executed independently from other elements in the stream. A stateful method such as `distinct` and `sorted` must be executed to take the entire stream into consideration. For example, to produce a result, the `distinct` method must consider all elements in the stream. Stateless methods are inherently parallelizable and can be executed in one pass in parallel. Stateful methods have to be executed in multiple passes in parallel.

stateless methods
stateful methods

Listing 30.3 gives an example to demonstrate the benefits of using parallel streams.

LISTING 30.3 ParallelStreamDemo.java

```
import java.util.Arrays;
import java.util.Random;
import java.util.stream.IntStream;

public class ParallelStreamDemo {
  public static void main(String[] args) {
    Random random = new Random();
    int[] list = random.ints(200_000_000).toArray();

    System.out.println("Number of processors: " +
      Runtime.getRuntime().availableProcessors());

    long startTime = System.currentTimeMillis();
    int[] list1 = IntStream.of(list).filter(e -> e > 0).sorted()
      .limit(5).toArray();
    System.out.println(Arrays.toString(list1));
    long endTime = System.currentTimeMillis();
    System.out.println("Sequential execution time is " +
      (endTime - startTime) + " milliseconds");

    startTime = System.currentTimeMillis();
    int[] list2 = IntStream.of(list).parallel().filter(e -> e > 0)
      .sorted().limit(5).toArray();
    System.out.println(Arrays.toString(list2));
    endTime = System.currentTimeMillis();
    System.out.println("Parallel execution time is " +
```

create an array
available processors
sequential stream
parallel stream

```
27        (endTime - startTime) + " milliseconds");
28    }
29  }
```

```
Number of processors: 8
[4, 9, 38, 42, 52]
Sequential execution time is 12362 milliseconds
[4, 9, 38, 42, 52]
Parallel execution time is 3448 milliseconds
```

The `Random` class introduced in Section 9.6.2 can be used to generate random numbers. You can use its `ints(n)` method to generate an `IntStream` consisting of `n` number of random `int` values (line 8). You can also use `ints(n, r1, r2)` to generate an `IntStream` with `n` element in the range from `r1` (inclusive) to `r2` (exclusive), use `doubles(n)` and `doubles(n, r1, r2)` to generate a `DoubleStream` of random floating-point numbers. Invoking the `toArray()` method on an `IntStream` (line 8) returns an array of `int` values from the stream. Recall that you can use underscores in an integer `200_000_000` to improve readability (see Section 2.10.1).

Invoking `Runtime.getRuntim()` returns a `Runtime` object (line 11). Invoking `Runtime` object's `availableProcessors()` returns the number of available processors for the JVM. In this case, the system has 8 processors.

An `IntStream` is created using `IntStream.of(list)` (line 14). The intermediate `filter(e -> e > 0)` method selects positive integers from the stream. The intermediate `sorted()` method sorts the filtered stream. The intermediate `limit(5)` method selects the first five integers in the sorted stream (line 15). Finally, the terminal method `toArray()` returns an array from the 5 integers in the stream. This is a sequential stream. To turn it into a parallel stream, simply invoke the `parallel()` method (line 22), which sets the stream for parallel execution. As you see from the sample run, the parallel execution is much faster than the sequential execution.

Several interesting questions arise.

lazy intermediate methods

1. The intermediate methods are lazy and are executed when the terminal method is initiated. This can be confirmed in the following code:

```
1    long startTime = System.currentTimeMillis();
2    IntStream stream = IntStream.of(list).filter(e -> e > 0).sorted()
3      .limit(5);
4    System.out.println("The time for the preceding method is " +
5      (System.currentTimeMillis() - startTime) + " milliseconds");
6    int[] list1 = stream.toArray();
7    System.out.println("The execution time is " +
8      (System.currentTimeMillis() - startTime) + " milliseconds");
```

   When you run the code, you will see that almost no time is spent on lines 2–3 because the intermediate methods are not executed yet. When the terminal method `toArray()` is invoked, all the methods for the stream pipeline are executed. So, the actual execution time for the stream pipeline is in line 6.

order of methods

2. Does the order of the intermediate methods in a stream pipeline matter? Yes, it matters. For example, if the methods `limit(5)` and `sorted()` are swapped, the result will be different. It also matters to the performance even though the result is the same. For example, if the `sorted()` method is placed before `filter(e -> e > 0)`, the result will be the same, but it would take more time to execute the stream because sorting a large number of elements takes more time to complete. Applying `filter` before `sorted` would eliminate roughly half of the elements for sorting.

parallel vs. sequential streams

3. Is a parallel stream always faster? Not necessarily. Parallel execution requires synchronization, which carries some overhead. If you replace `IntStream.of(list)` in lines 14

and 22 by `random.ints(200_000_000)`, the parallel stream in lines 22–23 takes longer time to execute than the sequential stream in lines 14–15. The reason is that the algorithm for generating a sequence of pseudo-random numbers is highly sequential. The overhead of a parallel stream is far greater than the time saving on parallel processing. So, you should test both sequential and parallel streams before choosing parallel streams for deployment.

4. When executing a stream method in parallel, the elements in the stream may be processed in any order. So, the following code may display the numbers in the stream in a random order:

order of parallel execution

```
IntStream.of(1, 2, 3, 4, 5).parallel()
  .forEach(e -> System.out.print(e + " "));
```

However, if it is executed sequentially, the numbers will be displayed as 1 2 3 4 5.

Check Point

30.4.1 What is a stateless method? What is a stateful method?

30.4.2 How do you create a parallel stream?

30.4.3 Suppose `names` is a set of strings, which of the following two streams is better?

```
Object[] s = set.parallelStream().filter(e -> e.length() > 3)
  .sorted().toArray();

Object[] s = set.parallelStream().sorted()
  .filter(e -> e.length() > 3).toArray();
```

30.4.4 What will be the output of the following code?

```
int[] values = {3, 4, 1, 5, 20, 1, 3, 3, 4, 6};
System.out.print("The values are ");
  IntStream.of(values)
    .forEach(e -> System.out.print(e + " "));
```

30.4.5 What will be the output of the following code?

```
int[] values = {3, 4, 1, 5, 20, 1, 3, 3, 4, 6};
System.out.print("The values are ");
  IntStream.of(values).parallel()
    .forEach(e -> System.out.print(e + " "));
```

30.4.6 Write a statement to obtain an array of `1000` random double values between `0.0` and `1.0`, excluding `1.0`.

30.5 Stream Reduction Using the `reduce` Method

You can use the `reduce` *method to reduce the elements in a stream into a single value.*

Key Point

Often you need to process all the elements in a collection to produce a summary value such as the sum, the maximum, or the minimum. For example, the following code obtains the sum of all elements in set `s`:

```
int total = 0;
for (int e: s) {
  total += e;
}
```

This is a simple code, but it specifies the exact steps on how to obtain the sum and it is highly sequential. The `reduce` method on a stream can be used to write the code in a high level for parallel execution.

reduction

A *reduction* takes the elements from a stream to produce a single value by repeated application of a binary operation such as addition, multiplication, or finding the maximum between two elements. Using reduction, you can write the code for finding the sum of all elements in a set as follows:

```
int sum = s.parallelStream().reduce(0, (e1, e2) -> e1 + e2);
```

Here, the `reduce` method takes two arguments. The first is an identity, that is, the starting value. The second argument is an object of the functional interface `IntBinaryOperator`. This interface contains the abstract method `applyAsInt(int e1, int e2)` that returns an `int` value from applying a binary operation. The preceding lambda expression in (a) is equivalent to the code using an anonymous inner class in (b).

```
reduce(0, e -> (e1, e2) -> e1 + e2)
```

(a) Using a lambda expression

```
reduce(0,
  new java.util.function.IntBinaryOperator() {
    public int applyAsInt(int e1, int e2) {
      return e1 + e2;
    }
  }
)
```

(b) Using an anonymous inner class

The preceding `reduce` method is semantically equivalent to an imperative code as follows:

```
int total = identity (i.e., 0, in this case);
for (int e: s) {
  total = applyAsInt(total, e);
}
```

The `reduce` method makes the code concise. Moreover, the code can be parallelizable, because multiple processors can simultaneously invoke the `applyAsInt` method on two integers repeatedly.

Using the `reduce` method, you can write the following code to return the maximum element in the set:

```
int result = s.parallelStream()
  .reduce(Integer.MIN_VALUE, (e1, e2) -> Math.max(e1, e2));
```

In fact, the `sum`, `max`, and `min` methods are implemented using the `reduce` method.

Listing 30.4 gives an example of using the `reduce` method.

LISTING 30.4 StreamReductionDemo.java

create an array

`forEach`

`reduce`

`mapToObj`

`reduce`

```
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class StreamReductionDemo {
  public static void main(String[] args) {
    int[] values = {3, 4, 1, 5, 20, 1, 3, 3, 4, 6};

    System.out.print("The values are ");
      IntStream.of(values).forEach(e -> System.out.print(e + " "));

    System.out.println("\nThe result of multiplying all values is " +
      IntStream.of(values).parallel().reduce(1, (e1, e2) -> e1 * e2));

    System.out.print("The values are " +
      IntStream.of(values).mapToObj(e -> e + "")
        .reduce((e1, e2) -> e1 + ", " + e2).get());

18      String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
19        "George", "Alan", "Stacy", "Michelle", "john"};
20      System.out.print("\nThe names are: ");
21      System.out.println(Stream.of(names)
22        .reduce((x, y) -> x + ", " + y).get());
23
24      System.out.print("Concat names: ");
25      System.out.println(Stream.of(names)
26        .reduce((x, y) -> x + y).get());
27
28      System.out.print("Total number of characters: ");
29      System.out.println(Stream.of(names)
30        .reduce((x, y) -> x + y).get().length());
31    }
32  }
```

`reduce`

```
The values are 3, 4, 1, 5, 20, 1, 3, 3, 4, 6
The result of multiplying all values is 259200
The values are 3, 4, 1, 5, 20, 1, 3, 3, 4, 6
The names are John, Peter, Susan, Kim, Jen, George, Alan, Stacy,
  Michelle, john
Concat names: JohnPeterSusanKimJenGeorgeAlanStacyMichellejohn
Total number of characters: 47
```

The program creates an array of `int` values (line 6). The stream pipeline creates an `IntStream` from the `int` array and invokes the `forEach` method to display each integer in the stream (line 9).

The program creates a parallel stream pipeline for the `int` array and applies the `reduce` method to obtain the product of `int` values in the stream (line 12).

The `mapToObj` method returns a stream of string objects from the `IntStream` (line 15). The `reduce` method can be called without an identity. In this case, it returns an object of `Optional<T>`. The `reduce` method in line 16 reduces the strings in the stream into one composite string that consists of all strings in the stream separated by commas.

The `reduce` method in line 26 combines all strings in the stream together into one long string. `Stream.of(names).reduce((x, y) -> x + y).get()` returns a string that concatenates all strings in the stream into one string, and invoking the `length()` method on the string returns the number of characters in the string (line 30).

Note `reduce((x, y) -> x + y)` in line 30 can be simplified using a method reference as `reduce(String::concat)`.

Check Point

30.5.1 Show the output of the following code:

```
int[] values = {1, 2, 3, 4};
System.out.println(IntStream.of(values)
  .reduce(0, (e1, e2) -> e1 + e2));
System.out.println(IntStream.of(values)
  .reduce(1, (e1, e2) -> e1 * e2));
System.out.println(IntStream.of(values).map(e -> e * e)
  .reduce(0, (e1, e2) -> e1 + e2));
System.out.println(IntStream.of(values).mapToObj(e -> "" + e)
  .reduce((e1, e2) -> e1 + " " + e2).get());
System.out.println(IntStream.of(values).mapToObj(e -> "" + e)
  .reduce((e1, e2) -> e1 + ", " + e2).get());
```

30.5.2 Show the output of the following code:

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}};
System.out.println(Stream.of(m)
  .map(e -> IntStream.of(e).reduce(1, (e1, e2) -> e1 * e2))
  .reduce(1, (e1, e2) -> e1 * e2));
```

30.5.3 Show the output of the following code:

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}, {1, 3}};
Stream.of(m).map(e -> IntStream.of(e))
  .reduce((e1, e2) -> IntStream.concat(e1, e2))
  .get().distinct()
  .forEach(e -> System.out.print(e + " "));
```

30.5.4 Show the output of the following code:

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}, {1, 3}};
System.out.println(
  Stream.of(m).map(e -> IntStream.of(e))
    .reduce((e1, e2) -> IntStream.concat(e1, e2))
    .get().distinct().mapToObj(e -> e + "")
    .reduce((e1, e2) -> e1 + ", " + e2).get());
```

30.6 Stream Reduction Using the `collect` Method

Key Point

You can use the `collect` method to reduce the elements in a stream into a mutable container.

In the preceding example, the `String`'s `concat` method is used in the `reduce` method for `Stream.of(names).reduce((x, y) -> x + y)`. This operation causes a new string to be created when concatenating two strings, which is very inefficient. A better approach is to use a `StringBuilder` and accumulate the result into a `StringBuilder`. This can be accomplished using the `collect` method.

The `collect` method collects the elements in a stream into a mutable container such as a `Collection` object using the following syntax:

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner)
```

The method takes three functional arguments: 1) a supplier function to construct a new instance of the result container, 2) an accumulator function to incorporate the elements from the stream to the result container, and 3) a combining function to merge the contents of one result container into another.

For example, to combine strings into a `StringBuilder`, you may write the following code using a `collect` method like this:

```
String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
  "George", "Alan", "Stacy", "Michelle", "john"};
StringBuilder sb = Stream.of(names).collect(() -> new StringBuilder(),
  (c, e) -> c.append(e), (c1, c2) -> c1.append(c2));
```

The lambda expression `() -> new StringBuilder()` creates a `StringBuilder` object for storing the result, which can be simplified using the method reference `StringBuilder::new`. The lambda expression `(c, e) -> c.append(e)` adds a string `e` to a `StringBuilder c`, which can be simplified using a method reference `StringBuilder::append`.

The lambda expression `(c1, c2) -> c1.append(c2)` merges the contents in `c2` into `c1`, which also can be simplified using a method reference `StringBuilder::append`. So, you can simplify the preceding statement as follows:

```
StringBuilder sb = Stream.of(names).collect(StringBuilder::new,
  StringBuilder::append, StringBuilder::append);
```

The sequential foreach loop implementation for this `collect` method might be as follows:

```
StringBuilder sb = new StringBuilder();
for (String s: Stream.of(names)) {
  sb.append(s);
}
```

Note that the combiner function `(c1, c2) -> c1.append(c2)` is not used in the sequential implementation. It is used when the stream pipeline is executed in parallel. When executing a `collect` method in parallel, multiple result `StringBuilder` are created and then merged using a combiner function. So, the purpose of the combiner function is for parallel processing.

Here is another example that creates an `ArrayList` from strings in the stream:

```
ArrayList<String> list = Stream.of(names).collect(ArrayList::new,
  ArrayList::add, ArrayList::addAll);
```

The supplier function is the `ArrayList` constructor. The accumulator is the `add` method that adds an element to the `ArrayList`. The combiner function merges an `ArrayList` into another `ArrayList`. The three arguments—supplier, accumulator, and combiner—are tightly coupled and are defined using standard methods. For simplicity, Java provides another `collect` method that takes an argument of the `Collector` type, called a *collector*. The `Collector` interface defines the methods for returning a supplier, an accumulator, and a combiner. You can use a static factory method `toList()` in the `Collectors` class to create an instance of the `Collector` interface. So, the preceding statement can be simplified using a standard collector as follows:

```
List<String> list = Stream.of(names).collect(Collectors.toList());
```

Listing 30.5 gives an example of using the `collect` methods and the `Collectors`' factory methods.

LISTING 30.5 `CollectDemo.java`

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectDemo {
  public static void main(String[] args) {
    String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
      "George", "Alan", "Stacy", "Michelle", "john"};
    System.out.println("The number of characters for all names: " +
      Stream.of(names).collect(StringBuilder::new,
        StringBuilder::append, StringBuilder::append).length());

    List<String> list = Stream.of(names).collect(ArrayList::new,
      ArrayList::add, ArrayList::addAll);
    System.out.println(list);
```

create an array

collect into `StringBuilder`

collect into `ArrayList`

`Collectors.toList()`

`Collectors.toSet()`

`Collectors.toMap()`

sum of integers

summary information

```
19
20      list = Stream.of(names).collect(Collectors.toList());
21      System.out.println(list);
22
23      Set<String> set = Stream.of(names).map(e -> e.toUpperCase()).
24        collect(Collectors.toSet());
25      System.out.println(set);
26
27      Map<String, Integer> map = Stream.of(names).collect(
28        Collectors.toMap(e -> e, e -> e.length()));
29      System.out.println(map);
30
31      System.out.println("The total number of characters is " +
32        Stream.of(names).
33          collect(Collectors.summingInt(e -> e.length())));
34
35      java.util.IntSummaryStatistics stats = Stream.of(names).
36        collect(Collectors.summarizingInt(e -> e.length()));
37      System.out.println("Max is " + stats.getMax());
38      System.out.println("Min is " + stats.getMin());
39      System.out.println("Average is " + stats.getAverage());
40    }
41  }
```

```
The number of characters for all names: 47
[John, Peter, Susan, Kim, Jen, George, Alan, Stacy, Michelle, john]
[John, Peter, Susan, Kim, Jen, George, Alan, Stacy, Michelle, john]
[JEN, GEORGE, ALAN, SUSAN, JOHN, PETER, MICHELLE, KIM, STACY]
{Michelle=8, Stacy=5, Jen=3, George=6, Susan=5, Alan=4, John=4,
  john=4, Peter=5, Kim=3}
The total number of characters is 47
Max is 8
Min is 3
Average is 4.7
```

The program creates an array of strings, **`names`** (lines 10–11). The **`collect`** method (line 13) specifies a supplier (**`StringBuilder::new`**) for creating a **`StringBuilder`**, an accumulator (**`StringBuilder::append`**) for adding a string to the **`StringBuilder`**, and a combiner (**`StringBuilder::append`**) for combining two **`StringBuilders`** (lines 13–14). The stream pipeline obtains a **`StringBuilder`** that contains all strings in the stream. The **`length()`** method returns the length of the characters in the **`StringBuilder`**.

The **`collect`** method (line 16) specifies a supplier (**`ArrayList::new`**) for creating an **`ArrayList`**, an accumulator (**`ArrayList::add`**) for adding a string to the **`ArrayList`**, and a combiner (**`ArrayList:addAll`**) for combining two **`ArrayLists`** (lines 16–17). The stream pipeline obtains an **`ArrayList`** that contains all strings in the stream. This statement is simplified using a standard collector **`Collectors.toList()`** for a list (line 20).

The program creates a string stream, maps each string to uppercase (line 23), and creates a set using the **`collect`** method with a standard collector **`Collectors.toSet()`** for a set (lines 24). There are two uppercase strings **`JOHN`** in the stream. Since a set contains no duplicate elements, only one **`JOHN`** is stored in the set.

The program creates a string stream and creates a map using the **`collect`** method with a standard collector for a map (lines 28). The key of the map is the string and the value is the length of the string. Note key must be unique. If two strings are identical in the stream, a runtime exception would occur.

The **`Collectors`** class also contains the method for returning collectors that produce summary information. For example, **`Collectors.summingInt`** produces the sum of integer values in the stream (line 33), and **`Collectors.summarizingInt`** produces an **`IntSummaryStatistics`** for the integer values in the stream (lines 35–36).

30.6.1 Show the output of the following code:

```
int[] values = {1, 2, 3, 4, 1};
List<Integer> list = IntStream.of(values).mapToObj(e -> e)
  .collect(Collectors.toList());
System.out.println(list);

Set<Integer> set = IntStream.of(values).mapToObj(e -> e)
  .collect(Collectors.toSet());
System.out.println(set);

Map<Integer, Integer> map = IntStream.of(values).distinct()
  .mapToObj(e -> e)
  .collect(Collectors.toMap(e -> e, e -> e.hashCode()));
System.out.println(map);

System.out.println(
  IntStream.of(values).mapToObj(e -> e)
    .collect(Collectors.summingInt(e -> e)));

System.out.println(
  IntStream.of(values).mapToObj(e -> e)
    .collect(Collectors.averagingDouble(e -> e)));
```

30.7 Grouping Elements Using the **groupingby** Collector

You can use the **`groupingBy`** *collector along with the* **`collect`** *method to collect the elements by groups.*

The elements in a stream can be divided into groups using the **`groupingBy`** collector and then applying aggregate collectors on each group. For example, you can group all the strings by their first letter and obtain the count of the elements in the group as follows:

```
String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
  "George", "Alan", "Stacy", "Steve", "john"};
Map<Character, Long> map = Stream.of(names).collect(
  Collectors.groupingBy(e -> e.charAt(0), Collectors.counting()));
```

group classifier
group processor

The first argument in the **`groupingBy`** method specifies the criteria for grouping, known as a *classifier*. The second argument specifies how the elements in a group are processed, known as a group *processor*. A processor is commonly a summary collector such as **`counting()`**. Using the **`groupingBy`** collector, the **`collect`** method returns a map with the classifier as the key. You may also specify a supplier in the **`groupingBy`** method such as the following:

```
Map<Character, Long> map = Stream.of(names).collect(
Collectors.groupingBy(e -> e.charAt(0),
  TreeMap::new, Collectors.counting()));
```

In this case, a tree map is used to store the map entries.

Listing 30.6 gives an example of using the **`groupingBy`** method.

LISTING 30.6 CollectGroupDemo.java

create an array

groupingBy

groupingBy

groupingBy

groupingby

Student class

```
import java.util.Map;
import java.util.TreeMap;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class CollectGroupDemo {
  public static void main(String[] args) {
    String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
      "George", "Alan", "Stacy", "Steve", "john"};

    Map<String, Long> map1 = Stream.of(names).
      map(e -> e.toUpperCase()).collect(
        Collectors.groupingBy(e -> e, Collectors.counting()));
    System.out.println(map1);

    Map<Character, Long> map2 = Stream.of(names).collect(
      Collectors.groupingBy(e -> e.charAt(0), TreeMap::new,
        Collectors.counting()));
    System.out.println(map2);

    int[] values = {2, 3, 4, 1, 2, 3, 2, 3, 4, 5, 1, 421};
    IntStream.of(values).mapToObj(e -> e).collect(
      Collectors.groupingBy(e -> e, TreeMap::new,
        Collectors.counting())).
      forEach((k, v) -> System.out.println(k + " occurs " + v +
        (v > 1 ? " times " : " time ")));

    MyStudent[] students = {new MyStudent("John", "Lu", "CS", 32, 78),
      new MyStudent("Susan", "Yao", "Math", 31, 85.4),
      new MyStudent("Kim", "Johnson", "CS", 30, 78.1)};

    System.out.printf("%10s%10s\n", "Department", "Average");
    Stream.of(students).collect(Collectors.
      groupingBy(MyStudent::getMajor, TreeMap::new,
        Collectors.averagingDouble(MyStudent::getScore))).
      forEach((k, v) -> System.out.printf("%10s%10.2f\n", k, v));
  }
}

class MyStudent {
  private String firstName;
  private String lastName;
  private String major;
  private int age;
  private double score;

  public MyStudent(String firstName, String lastName, String major,
      int age, double score) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.major = major;
    this.age = age;
    this.score = score;
  }

  public String getFirstName() {
    return firstName;
59     }
60
61     public String getLastName() {
62       return lastName;
63     }
64
65     public String getMajor() {
66       return major;
67     }
68
69     public int getAge() {
70       return age;
71     }
72
73     public double getScore() {
74       return score;
75     }
76   }
```

```
{JEN=1, ALAN=1, GEORGE=1, SUSAN=1, JOHN=2, STEVE=1, PETER=1, STACY=1,
  KIM=1}
{A=1, G=1, J=2, K=1, P=1, S=3, j=1}
1 occurs 2 times
2 occurs 3 times
3 occurs 3 times
4 occurs 2 times
5 occurs 1 time
421 occurs 1 time
Department   Average
        CS     78.05
      Math     85.40
```

The program creates a string stream, maps its elements to uppercase (lines 12–13), and collects the elements into a map with the string as the key and the occurrence of the string as the value (line 14). Note the `counting()` collector uses a value of the `Long` type. So, the `Map` is declared `Map<String, Long>`.

The program creates a string stream and collects the elements into a map with the first character in the string as the key and the occurrence of the first character as the value (lines 17–18). The key and value entries are stored in a `TreeMap`.

The program creates an `int` array (line 22). A stream is created from this array and the elements are mapped to `Integer` objects using the `mapToObj` method (line 23). The `collect` method returns a map with the `Integer` value as the key and the occurrence of the integer as the value (lines 24–25). The `forEach` method displays the key and value entries. Note the `collect` method is a terminal method. It returns an instance of `TreeMap` in this case. `TreeMap` has the `forEach` method for performing an action on each element in the collection.

The program creates an array of `Student` objects (lines 29–31). The `Student` class is defined with properties `firstName`, `lastName`, `major`, `age`, and `score` in lines 41–76. The program creates a stream for the array and the `collect` method groups the students by their major and returns a map with the major as the key and the average scores for the group as the value (lines 34-36). The method reference `MyStudent::getMajor` is used to specify the group by classifier. The method reference `TreeMap::new` specifies a supplier for the result map. The method reference `MyStudent:getScore` specifies the value for averaging.

Check Point

30.7.1 Show the output of the following code:

```
int[] values = {1, 2, 2, 3, 4, 2, 1};
IntStream.of(values).mapToObj(e -> e).collect(
  Collectors.groupingBy(e -> e, TreeMap::new,
    Collectors.counting())).
  forEach((k, v) -> System.out.println(k + " occurs " + v
    + (v > 1 ? " times " : " time ")));

IntStream.of(values).mapToObj(e -> e).collect(
  Collectors.groupingBy(e -> e, TreeMap::new,
    Collectors.summingInt(e -> e))).
  forEach((k, v) -> System.out.println(k + ": " + v));

MyStudent[] students = {
  new MyStudent("John", "Johnson", "CS", 23, 89.2),
  new MyStudent("Susan", "Johnson", "Math", 21, 89.1),
  new MyStudent("John", "Peterson", "CS", 21, 92.3),
  new MyStudent("Kim", "Yao", "Math", 22, 87.3),
  new MyStudent("Jeff", "Johnson", "CS", 23, 78.5)};

Stream.of(students)
  .sorted(Comparator.comparing(MyStudent::getLastName)
    .thenComparing(MyStudent::getFirstName))
  .forEach(e -> System.out.println(e.getLastName() + ", " +
                e.getFirstName()));

Stream.of(students).collect(Collectors.
  groupingBy(MyStudent::getAge, TreeMap::new,
    Collectors.averagingDouble(MyStudent::getScore))).
  forEach((k, v) -> System.out.printf("%10s%10.2f\n", k, v));
```

30.8 Case Studies

Key Point

Many programs for processing arrays and collections can now be simplified and run faster using aggregate methods on streams.

You can write programs without using streams. However, using streams enables you to write shorter and simpler programs that can be executed faster in parallel by utilizing multiple processors. Many of the programs that involve arrays and collections in the early chapters can be simplified using streams. This section presents several case studies.

30.8.1 Case Study: Analyzing Numbers

Section 7.3 gives a program that prompts the user to enter values, obtains their average, and displays the number of values greater than the average. The program can be simplified using a `DoubleStream` as shown in Listing 30.7.

Listing 30.7 AnalyzeNumbersUsingStream.java

```
import java.util.stream.*;

public class AnalyzeNumbersUsingStream {
  public static void main(String[] args) {
    java.util.Scanner input = new java.util.Scanner(System.in);
    System.out.print("Enter the number of items: ");
    int n = input.nextInt();
    double[] numbers = new double[n];
    double sum = 0;
```

create array

```
10
11     System.out.print("Enter the numbers: ");
12     for (int i = 0; i < n; i++) {
13       numbers[i] = input.nextDouble();
14     }
15
16     double average = DoubleStream.of(numbers).average().getAsDouble();
17     System.out.println("Average is " + average);
18     System.out.println("Number of elements above the average is "
19       + DoubleStream.of(numbers).filter(e -> e > average).count());
20   }
21 }
```

store number in array

get average

above average?

```
Enter the number of items: 10 ↵Enter
Enter the numbers: 3.4 5 6 1 6.5 7.8 3.5 8.5 6.3 9.5 ↵Enter
Average is 5.75
Number of elements above the average is 6
```

The program obtains the input from the user and stores the values in an array (lines 8–14), and obtains the average of the values using a stream (line 16), and finds the number of values greater than the average using a filtered stream (line 19).

30.8.2 Case Study: Counting the Occurrences of Each Letter

Listing 7.4, CountLettersInArrays.java, gives a program that randomly generates 100 lowercase letters and counts the occurrences of each letter.

The program can be simplified using a `Stream` as shown in Listing 30.8.

Listing 30.8 CountLettersUsingStream.java

```
import java.util.Random;
import java.util.TreeMap;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CountLettersUsingStream {
  private static int count = 0;

  public static void main(String[] args) {
    Random random = new Random();
    Object[] chars = random.ints(100, (int)'a', (int)'z' + 1).
      mapToObj(e -> (char)e).toArray();

    System.out.println("The lowercase letters are:");
    Stream.of(chars).forEach(e -> {
      System.out.print(e + (++count % 20 == 0 ? "\n" : " "));
    });

    count = 0; // Reset the count for columns
    System.out.println("\nThe occurrences of each letter are:");
    Stream.of(chars).collect(Collectors.groupingBy(e -> e,
      TreeMap::new, Collectors.counting())).forEach((k, v) -> {
        System.out.print(v + " " + k
          + (++count % 10 == 0 ? "\n" : " "));
      });
  }
}
```

create array

display array

count occurrence

```
The lowercase letters are:
e y l s r i b k j v j h a b z n w b t v
s c c k r d w a m p w v u n q a m p l o
a z g d e g f i n d x m z o u l o z j v
h w i w n t g x w c d o t x h y v z y z
q e a m f w p g u q t r e n n w f c r f

The occurrences of each letter are:
5 a 3 b 4 c 4 d 4 e 4 f 4 g 3 h 3 i 3 j
2 k 3 l 4 m 6 n 4 o 3 p 3 q 4 r 2 s 4 t
3 u 5 v 8 w 3 x 3 y 6 z
```

The program generates a stream of 100 random integers. These integers are in the range between **`(char)'a'`** and **`(char)'z'`** (line 11). They are ASCII code for the lowercase letters. The **`mapToObj`** method maps the integers to their corresponding lowercase letters (line 12). The **`toArray()`** method returns an array consisting of these lowercase letters.

The program creates a stream of lowercase letters (line 15) and uses the **`forEach`** method to display each letter (line 16). The letters are displayed 20 per line. The static variable **`count`** is used to count the letters printed.

The program resets the count to **`0`** (line 19) , creates a stream of lowercase letters (line 21), returns a map with lowercase letters as the key and the occurrences of each letter as the value (lines 21–22), and invokes the **`forEach`** method to display each key and value 10 per line (lines 23–24).

The code has 66 lines in Listing 7.4. The new code has only 27 lines, which greatly simplified coding. Furthermore, using streams is more efficient.

30.8.3 Case Study: Counting the Occurrences of Each Letter in a String

The preceding example randomly generates lowercase letters and counts the occurrence of each letter. This example counts the occurrence of each letter in a string. The program given in Listing 30.9 prompts the user to enter the string, converts all letters to uppercase, and displays the count of each letter in the string.

LISTING 30.9 `CountOccurrenceOfLettersInAString.java`

```java
import java.util.*;
import java.util.stream.Stream;
import java.util.stream.Collectors;

public class CountOccurrenceOfLettersInAString {
  private static int count = 0;

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a string: ");
    String s = input.nextLine();

    count = 0; // Reset the count for columns
    System.out.println("The occurrences of each letter are:");
    Stream.of(toCharacterArray(s.toCharArray()))
      .filter(ch -> Character.isLetter(ch))
      .map(ch -> Character.toUpperCase(ch))
      .collect(Collectors.groupingBy(e -> e,
```

read a string

convert `char[]` to `Character[]`

select letters

map to uppercase

collect and count

```
19            TreeMap::new, Collectors.counting()))
20            .forEach((k, v) -> { System.out.print(v + " " + k
21              + (++count %  10 == 0 ?  "\n" :  " "));
22            });
23    }
24
25    public static Character[] toCharacterArray(char[] list) {
26      Character[] result = new Character[list.length];
27      for (int i = 0; i < result.length; i++) {
28        result[i] = list[i];
29      }
30      return result;
31    }
32  }
```

display value and key

```
Enter a string: Welcome to JavaAA ↵Enter
The occurrences of each letter are:
4 A 1 C 2 E 1 J 1 L 1 M 2 O 1 T 1 V 1 W
```

The program reads a string **`s`** and obtains an array of **`char`** from the string by invoking **`s.toCharArray()`**. To create a stream of characters, you need to convert **`char[]`** into **`Character[]`**. So, the program defines the **`toCharacterArray`** method (lines 25–31) for obtaining a **`Character[]`** from **`char[]`**.

`char[] to Character[]`

The program creates a stream of **`Character`** object (line 15), eliminates nonletters from the stream using the **`filter`** method (line 16), coverts all letters to uppercase using the **`map`** method (line 17), and obtains a **`TreeMap`** using the **`collect`** method (lines 18–19). In the **`TreeMap`**, the key is the letter and the value is the count for the letter. The **`forEach`** method (lines 20–21) in **`TreeMap`** is used to display the value and key.

30.8.3 Case Study: Processing All Elements in a Two-Dimensional Array

You can create a stream from a one-dimensional array. Can you create a stream for processing two-dimensional arrays? Listing 30.10 gives an example of processing two-dimensional arrays using streams.

LISTING 30.10 `TwoDimensionalArrayStream.java`

```
import java.util.IntSummaryStatistics;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class TwoDimensionalArrayStream {
  private static int i = 0;
  public static void main(String[] args) {
    int[][] m = {{1, 2}, {3, 4}, {4, 5}, {1, 3}};

    int[] list = Stream.of(m).map(e -> IntStream.of(e)).
      reduce((e1, e2) -> IntStream.concat(e1, e2)).get().toArray();

    IntSummaryStatistics stats =
      IntStream.of(list).summaryStatistics();
    System.out.println("Max: " + stats.getMax());
    System.out.println("Min: " + stats.getMin());
```

create an array

reduce to one-dimensional

obtain statistical information

sum of each row

```
17      System.out.println("Sum: " + stats.getSum());
18      System.out.println("Average: " + stats.getAverage());
19
20      System.out.println("Sum of row ");
21      Stream.of(m).mapToInt(e -> IntStream.of(e).sum())
22       .forEach(e ->
23        System.out.println("Sum of row " + i++ + ": " + e));
24    }
25  }
```

```
Max: 5
Min: 1
Sum: 23
Average: 2.875
Sum of row
Sum of row 0: 3
Sum of row 1: 7
Sum of row 2: 9
Sum of row 3: 4
```

The program creates a two-dimensional array **m** in line 8. Invoking **Stream.of(m)** creates a stream consisting of rows as elements (line 10). The **map** method maps each row to an **IntStream**. The **reduce** method concatenates these streams into one large stream (line 11). This large stream now contains all the elements in **m**. Invoking **toArray()** from the stream returns an array consisting of all the integers in the stream (line 11).

The program obtains a statistical summary for an **IntStream** created from the array (lines 13–14) and displays the maximum, minimum, sum, and average of the integers in the stream (lines 15–18).

Finally, the program creates a stream from **m** that consists of rows in **m** (line 21). Each row is mapped to an **int** value using the **mapToInt** method (line 21). The **int** value is the sum of the elements in the row. The **forEach** method displays the sum for each row (lines 22–23).

30.8.4 Case Study: Finding the Directory Size

Listing 18.7 gives a recursive program that finds the size of a directory. The size of a directory is the sum of the sizes of all the files in the directory. The program can be implemented using streams as shown in Listing 30.11.

LISTING 30.11 DirectorySizeStream.java

invoke method

```
import java.io.File;
import java.nio.file.Files;
import java.util.Scanner;

public class DirectorySizeStream {
  public static void main(String[] args)  throws Exception {
    // Prompt the user to enter a directory or a file
    System.out.print("Enter a directory or a file: ");
    Scanner input = new Scanner(System.in);
    String directory = input.nextLine();

    // Display the size
    System.out.println(getSize(new File(directory)) + " bytes");
  }

16    public static long getSize(File file) {
17      if (file.isFile()) {
18        return file.length();
19      }
20      else {
21        try {
22          return Files.list(file.toPath()).parallel().
23            mapToLong(e -> getSize(e.toFile())).sum();
24        } catch (Exception ex) {
25          return 0;
26        }
27      }
28    }
29  }
```

getSize method
is file?
return file length

get subpaths
recursive call

```
Enter a directory or a file: c:\book ↵Enter
48619631 bytes
```

```
Enter a directory or a file: c:\book\Welcome.java ↵Enter
172 bytes
```

```
Enter a directory or a file: c:\book\NonExistentFile ↵Enter
0 bytes
```

The program prompts the user to enter a file or a directory name (lines 8–10) and invokes the `getSize(File file)` method to return the size of the file or a directory (line 13). The `File` object can be a directory or a file. If it is a file, the `getSize` method returns the size of the file (lines 17–19). If it is a directory, invoking `Files.list(file.toPath())` returns a stream consisting of the `Path` objects. Each `Path` object represents a subpath in the directory (line 22). Since JDK 1.8, new methods are added in some existing classes to return streams. The `Files` class now has the static `list(Path)` method that returns a stream of subpaths in the path. For each subpath `e`, the `mapToLong` method maps `e` into the size of `e` by invoking `getSize(e)`. Finally, the terminal `sum()` method returns the size of the whole directory.

Both `getSize` methods in Listing 18.7 and Listing 30.9 are recursive. The `getSize` method in Listing 30.9 can be executed in a parallel stream. So, Listing 30.9 has better performance.

30.8.5 Case Study: Counting Keywords

Listing 21.7 gives a program to count the keywords in a Java source file. The program reads the words from a text file and tests if the word is a keyword. You can rewrite the code using streams as shown in Listing 30.12.

LISTING 30.12 `CountKeywordStream.java`

```
import java.util.*;
import java.io.*;
import java.nio.file.Files;
import java.util.stream.Stream;

public class CountKeywordStream {
  public static void main(String[] args) throws Exception {
    Scanner input = new Scanner(System.in);
```

enter a filename

file exists?

count keywords

keywords

keyword set

count keyword

```
 9      System.out.print("Enter a Java source file: ");
10      String filename = input.nextLine();
11
12      File file = new File(filename);
13      if (file.exists()) {
14        System.out.println("The number of keywords in " + filename
15          + " is " + countKeywords(file));
16      }
17      else {
18        System.out.println("File " + filename + " does not exist");
19      }
20    }
21
22    public static long countKeywords(File file) throws Exception {
23      // Array of all Java keywords + true, false and null
24      String[] keywordString = {"abstract", "assert", "boolean",
25          "break", "byte", "case", "catch", "char", "class", "const",
26          "continue", "default", "do", "double", "else", "enum",
27          "extends", "for", "final", "finally", "float", "goto",
28          "if", "implements", "import", "instanceof", "int",
29          "interface", "long", "native", "new", "package", "private",
30          "protected", "public", "return", "short", "static",
31          "strictfp", "super", "switch", "synchronized", "this",
32          "throw", "throws", "transient", "try", "void", "volatile",
33          "while", "true", "false", "null"};
34
35      Set<String> keywordSet =
36        new HashSet<>(Arrays.asList(keywordString));
37
38      return Files.lines(file.toPath()).parallel().mapToLong(line ->
39        Stream.of(line.split("[\\s++]")).
40          filter(word -> keywordSet.contains(word)).count()).sum();
41    }
42  }
```

```
Enter a Java source file: c:\Welcome.java ↵ Enter
The number of keywords in c:\Welcome.java is 5
```

```
Enter a Java source file: c:\TTT.java ↵ Enter
File c:\TTT.java does not exist
```

The program prompts the user to enter a filename (lines 9–10). If the file exists, it invokes `conutKeywords(file)` to return the number of keywords in the file (line 15). The keywords are stored in a set `keywordSet` (lines 35–36). Invoking `Files.lines(file.toPath())` returns a streams of lines form the file (line 38). Each line in the stream is mapped to a `long` value that counts the number of the keywords in the line. The line is split into an array of words using `line.split("[\\s++]")` and this array is used to create a stream (line 39). The `filter` method is applied to select the keyword from the stream. Invoking `count()` returns the number of the keywords in a line. The `sum()` method returns the total number of keywords in all lines (line 40).

The real benefit of using parallel streams in this example is for improving performance (line 38).

30.8.6 Case Study: Occurrences of Words

Listing 21.9 gives a program that counts the occurrences of words in a text. You can rewrite the code using streams as shown in Listing 30.13.

LISTING 30.13 `CountOccurrenceOfWordsStream.java`

```
 1  import java.util.*;
 2  import java.util.stream.Collectors;
 3  import java.util.stream.Stream;
 4
 5  public class CountOccurrenceOfWordsStream {
 6    public static void main(String[] args) {
 7      // Set text in a string
 8      String text = "Good morning. Have a good class. "
 9        + "Have a good visit. Have fun!";
10
11      Stream.of(text.split("[\\s+\\p{P}]")).parallel()
12        .filter(e -> e.length() > 0).collect(
13          Collectors.groupingBy(String::toLowerCase, TreeMap::new,
14            Collectors.counting()))
15        .forEach((k, v) -> System.out.println(k + " " + v));
16    }
17  }
```

text

split into words

filter empty words

group by words

display counts

```
a          2
class      1
fun        1
good       3
have       3
morning    1
visit      1
```

The text is split into words using a whitespace `\s` or punctuation `\p{P}` as a delimiter (line 11) and a stream is created for the words. The `filter` method is used to select nonempty words (line 12). The `collect` method groups the words by converting each into lowercase and returns a `TreeMap` with the lowercase words as the key and their count as the value (lines 13–14). The `forEach` method displays the key and its value (line 15).

The code in Listing 30.11 is about half in size as the code in Listing 21.9. The new program greatly simplifies coding and improves performance.

Check Point

30.8.1 Can the following code be used to replace line 19 in Listing 30.7?

```
DoubleStream.of(numbers).filter(e -> e >
  DoubleStream.of(numbers).average()).count());
```

30.8.2 Can the following code be used to replace lines 15–16 in Listing 30.8?

```
Stream.of(chars).forEach(e -> {
  int count = 0;
  System.out.print(e + (++count % 20 == 0 ? "\n" : " ")); });
```

30.8.3 Show the output of the following code:

```
String s = "ABC";
Stream.of(s.toCharArray()).forEach(ch ->
  System.out.println(ch));
```

30.8.4 Show the output of the following code (The `toCharacterArray` method is presented in Listing 30.9.):

```
String s = "ABC";
Stream.of(toCharacterArray(s.toCharArray())).forEach(ch ->
  System.out.println(ch));
```

30.8.5 Write the code to obtain a one-dimensional array `list` of strings from a two-dimensional array `matrix` of strings.

CHAPTER SUMMARY

1. Java 8 introduces aggregate methods on collection streams to simplify coding and improve performance.
2. A stream pipeline creates a stream from a data source, consists of zero or more intermediate methods (`skip`, `limit`, `filter`, `distinct`, `sorted`, `map`, and `mapToInt`), and a final terminal method (`count`, `sum`, `average`, `max`,`min`, `forEach`, `findFirst`, `firstAny`, `anyMatch`, `allMatch`, `noneMatch`, and `toArray`).
3. The execution of a stream is lazy, which means that the methods in the stream are not executed until the final terminal method is initiated.
4. The streams are transient objects. They are destroyed once the terminal method is executed.
5. The `Stream<T>` class defines the streams for a sequence of objects of the `T` type. `IntStream`, `LongStream`, and `DoubleStream` are the streams for a sequence of primitive `int`, `long`, and `double` values.
6. An important benefit of using streams is for performance. Streams can be executed in parallel mode to take advantages of multi-core architecture. You can switch a stream into a parallel or sequential mode by invoking the `parallel()` or `sequential()` method.
7. You can use the `reduce` method to reduce a stream into a single value and use the `collect` method to place the elements in the stream into a collection.
8. You can use the `groupingBy` collector to group the elements in the stream and apply aggregate methods for the elements in the group.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

30.1 (*Assign grades*) Rewrite Programming Exercise 7.1 using streams.

30.2 (*Count occurrence of numbers*) Rewrite Programming Exercise 7.3 using streams.

30.3 (*Analyze scores*) Rewrite Programming Exercise 7.4 using streams.

30.4 (*Print distinct numbers*) Rewrite Programming Exercise 7.5 using streams. Display the numbers in increasing order.

30.5 (*Count single digits*) Rewrite Programming Exercise 7.7 using streams.

30.6 (*Average an array*) Rewrite Programming Exercise 7.8 using streams.

30.7 (*Find the smallest element*) Rewrite Programming Exercise 7.9 using streams.

30.8 (*Eliminate duplicates*) Rewrite Programming Exercise 7.15 using streams and sort the elements in the new array in increasing order.

30.9 (*Sort students*) Rewrite Programming Exercise 7.17 using streams. Define a class named `Student` with data fields `name` and `score` and their getter methods. Store each student in a `Student` object.

30.10 (*Convert binary to decimal*) Write a program that prompts the user to enter a binary number in string and displays its decimal value. Use `Stream`'s `reduce` method to convert a binary number to decimal.

30.11 (*Convert hex to decimal*) Write a program that prompts the user to enter a hex number in string and displays its decimal value. Use `Stream`'s `reduce` method to convert a hex number to decimal.

30.12 (*Sum the digits in an integer*) Rewrite Programming Exercise 6.2 using streams.

30.13 (*Count the letters in a string*) Rewrite Programming Exercise 6.20 using streams.

30.14 (*Occurrences of a specified character*) Rewrite Programming Exercise 6.23 using streams.

30.15 (*Display words in ascending alphabetical order*) Rewrite Programming Exercise 20.1 using streams.

30.16 (*Distinct scores*) Use streams to write a program that displays the distinct scores in the `scores` array in Section 8.8. Display the scores in increasing order, separated by exactly one space, five numbers per line.

30.17 (*Count consonants and vowels*) Rewrite Programming Exercise 21.4 using streams.

30.18 (*Count the occurrences of words in a text file*) Rewrite Programming Exercise 21.8 using streams.

30.19 (*Summary information*) Suppose the file test.txt contains floating-point numbers separated by spaces. Write a program to obtain the sum, average, maximum, and minimum of the numbers.

APPENDIXES

Appendix A
Java Keywords and Reserved Words

Appendix B
The ASCII Character Set

Appendix C
Operator Precedence Chart

Appendix D
Java Modifiers

Appendix E
Special Floating-Point Values

Appendix F
Number Systems

Appendix G
Bitwise Operations

Appendix H
Regular Expressions

Appendix I
Enumerated Types

Appendix J
The Big-O, Big-Omega, and Big-Theta Notations

APPENDIX A

Java Keywords and Reserved Words

Keywords have special meaning in Java and are part of the syntax. Reserved words are the words that cannot be used as identifiers. Keywords are reserved words. The following 50 keywords are reserved for use by the Java language:

`abstract`
`assert`
`boolean`
`break`
`byte`
`case`
`catch`
`char`
`class`
`const`
`continue`
`default`
`do`
`double`
`else`
`enum`
`extends`
`final`
`finally`
`float`
`for`
`goto`
`if`
`implements`
`import`
`instanceof`
`int`
`interface`
`long`
`native`
`new`
`package`
`private`
`protected`
`public`
`return`
`short`
`static`
`strictfp`*
`super`
`switch`
`synchronized`
`this`
`throw`
`throws`
`transient`
`try`
`void`
`volatile`
`while`

The keywords `goto` and `const` are C++ keywords reserved, but not currently used in Java. This enables Java compilers to identify them and to produce better error messages if they appear in Java programs.

The literal values `true`, `false`, and `null` are reserved words, but not keywords. You cannot use them as identifiers.

In the code listing, we use the keyword color for `true`, `false`, and `null` to be consistent with their coloring in Java IDEs.

---

*The `strictfp` keyword is a modifier for a method or class that enables it to use strict floating-point calculations. Floating-point arithmetic can be executed in one of two modes: *strict* or *nonstrict*. The strict mode guarantees that the evaluation result is the same on all Java Virtual Machine implementations. The nonstrict mode allows intermediate results from calculations to be stored in an extended format different from the standard IEEE floating-point number format. The extended format is machine dependent and enables code to be executed faster. However, when you execute the code using the nonstrict mode on different JVMs, you may not always get precisely the same results. By default, the nonstrict mode is used for floating-point calculations. To use the strict mode in a method or a class, add the `strictfp` keyword in the method or the class declaration. Strict floating-point may give you slightly better precision than nonstrict floating-point, but the distinction will only affect some applications. Strictness is not inherited; that is, the presence of `strictfp` on a class or interface declaration does not cause extended classes or interfaces to be strict.

APPENDIX B

The ASCII Character Set

Tables B.1 and B.2 show ASCII characters and their respective decimal and hexadecimal codes. The decimal or hexadecimal code of a character is a combination of its row index and column index. For example, in Table B.1, the letter A is at row 6 and column 5, so its decimal equivalent is 65; in Table B.2, letter A is at row 4 and column 1, so its hexadecimal equivalent is 41.

TABLE B.1 ASCII Character Set in the Decimal Index

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	”	#	$	%	&	’
4	(	)	*	+	,	–	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	∧	–	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

TABLE B.2 ASCII Character Set in the Hexadecimal Index

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	ff	cr	so	si
1	dle	dcl	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2	sp	!	”	#	$	%	&	’	(	)	*	+	,	–	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	∧	–
6	’	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del

APPENDIX C

Operator Precedence Chart

The operators are shown in decreasing order of precedence from top to bottom. Operators in the same group have the same precedence, and their associativity is shown in the table.

Operator	*Name*	*Associativity*
`()`	Parentheses	Left to right
`()`	Function call	Left to right
`[]`	Array subscript	Left to right
`.`	Object member access	Left to right
`++`	Postincrement	Left to right
`--`	Postdecrement	Left to right
`++`	Preincrement	Right to left
`--`	Predecrement	Right to left
`+`	Unary plus	Right to left
`-`	Unary minus	Right to left
`!`	Unary logical negation	Right to left
`(type)`	Unary casting	Right to left
`new`	Creating object	Right to left
`*`	Multiplication	Left to right
`/`	Division	Left to right
`%`	Remainder	Left to right
`+`	Addition	Left to right
`-`	Subtraction	Left to right
`<<`	Left shift	Left to right
`>>`	Right shift with sign extension	Left to right
`>>>`	Right shift with zero extension	Left to right
`<`	Less than	Left to right
`<=`	Less than or equal to	Left to right
`>`	Greater than	Left to right
`>=`	Greater than or equal to	Left to right
`instanceof`	Checking object type	Left to right

Operator	*Name*	*Associativity*
`==`	Equal comparison	Left to right
`!=`	Not equal	Left to right
`&`	(Unconditional AND)	Left to right
`^`	(Exclusive OR)	Left to right
`|`	(Unconditional OR)	Left to right
`&&`	Conditional AND	Left to right
`||`	Conditional OR	Left to right
`?:`	Ternary condition	Right to left
`=`	Assignment	Right to left
`+=`	Addition assignment	Right to left
`-=`	Subtraction assignment	Right to left
`*=`	Multiplication assignment	Right to left
`/=`	Division assignment	Right to left
`%=`	Remainder assignment	Right to left

APPENDIX D

Java Modifiers

Modifiers are used on classes and class members (constructors, methods, data, and class-level blocks), but the `final` modifier can also be used on local variables in a method. A modifier that can be applied to a class is called a *class modifier*. A modifier that can be applied to a method is called a *method modifier*. A modifier that can be applied to a data field is called a *data modifier*. A modifier that can be applied to a class-level block is called a *block modifier*. The following table gives a summary of the Java modifiers.

Modifier	*Class*	*Constructor*	*Method*	*Data*	*Block*	*Explanation*
`(blank)`*	√	√	√	√	√	A class, constructor, method, or data field is visible in this package.
`public`	√	√	√	√		A class, constructor, method, or data field is visible to all the programs in any package.
`private`		√	√	√		A constructor, method, or data field is only visible in this class.
`protected`		√	√	√		A constructor, method, or data field is visible in this package and in subclasses of this class in any package.
`static`			√	√	√	Define a class method, a class data field, or a static initialization block.
`final`	√		√	√		A final class cannot be extended. A final method cannot be modified in a subclass. A final data field is a constant.
`abstract`	√		√			An abstract class must be extended. An abstract method must be implemented in a concrete subclass.
`native`			√			A native method indicates that the method is implemented using a language other than Java.

* (blank) means no modifiers are used. For example: `class Test {}`

Modifier	*Class*	*Constructor*	*Method*	*Data*	*Block*	*Explanation*
`synchronized`			√		√	Only one thread at a time can execute this method.
`strictfp`	√		√			Use strict floating-point calculations to guarantee that the evaluation result is the same on all JVMs.
`transient`				√		Mark a nonserializable instance data field.

The modifiers `public`, `private`, and `protected` are known as *visibility* or *accessibility modifiers* because they specify how classes and class members are accessed.

The modifiers `public`, `private`, `protected`, `static`, `final`, and `abstract` can also be applied to inner classes.

Java 8 introduced the `default` modifier for declaring a default method in an interface. A default method provides a default implementation for the method in the interface.

APPENDIX E

Special Floating-Point Values

Dividing an integer by zero is invalid and throws `ArithmeticException`, but dividing a floating-point value by zero does not cause an exception. Floating-point arithmetic can overflow to infinity if the result of the operation is too large for a `double` or a `float`, or underflow to zero if the result is too small for a `double` or a `float`. Java provides the special floating-point values `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, and `NaN` (Not a Number) to denote these results. These values are defined as special constants in the `Float` class and the `Double` class.

If a positive floating-point number is divided by zero, the result is `POSITIVE_INFINITY`. If a negative floating-point number is divided by zero, the result is `NEGATIVE_INFINITY`. If a floating-point zero is divided by zero, the result is `NaN`, which means that the result is undefined mathematically. The string representations of these three values are `Infinity`, `-Infinity`, and `NaN`. For example,

```
System.out.print(1.0 / 0); // Print Infinity
System.out.print(–1.0 / 0); // Print –Infinity
System.out.print(0.0 / 0); // Print NaN
```

These special values can also be used as operands in computations. For example, a number divided by `POSITIVE_INFINITY` yields a positive zero. Table E.1 summarizes various combinations of the `/`, `*`, `%`, `+`, and `–` operators.

TABLE E.1 Special Floating-Point Values

x	*y*	*x/y*	*x*y*	*x%y*	*x + y*	*x − y*
`Finite`	`±0.0`	`±infinity`	`±0.0`	`NaN`	`Finite`	`Finite`
`Finite`	`±infinity`	`±0.0`	`±0.0`	`x`	`±infinity`	`infinity`
`±0.0`	`±0.0`	`NaN`	`±0.0`	`NaN`	`±0.0`	`±0.0`
`±infinity`	`Finite`	`±infinity`	`±0.0`	`NaN`	`±infinity`	`±infinity`
`±infinity`	`±infinity`	`NaN`	`±0.0`	`NaN`	`±infinity`	`infinity`
`±0.0`	`±infinity`	`±0.0`	`NaN`	`±0.0`	`±infinity`	`±0.0`
`NaN`	`Any`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`
`Any`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`

Note

If one of the operands is NaN, the result is NaN.

APPENDIX F

Number Systems

F.1 Introduction

Computers use binary numbers internally, because computers are made naturally to store and process 0s and 1s. The binary number system has two digits, 0 and 1. A number or character is stored as a sequence of 0s and 1s. Each 0 or 1 is called a *bit* (binary digit).

base radix

In our daily life, we use decimal numbers. When we write a number such as 20 in a program, it is assumed to be a decimal number. Internally, computer software is used to convert decimal numbers into binary numbers, and vice versa.

decimal numbers

We write computer programs using decimal numbers. However, to deal with an operating system, we need to reach down to the "machine level" by using binary numbers. Binary numbers tend to be very long and cumbersome. Often hexadecimal numbers are used to abbreviate them, with each hexadecimal digit representing four binary digits. The hexadecimal number system has 16 digits: 0–9 and A–F. The letters A, B, C, D, E, and F correspond to the decimal numbers 10, 11, 12, 13, 14, and 15.

hexadecimal number

The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. A decimal number is represented by a sequence of one or more of these digits. The value that each digit represents depends on its position, which denotes an integral power of 10. For example, the digits 7, 4, 2, and 3 in decimal number 7423 represent 7000, 400, 20, and 3, respectively, as shown below:

$$\begin{array}{|c|c|c|c|}\hline 7 & 4 & 2 & 3 \\ \hline \end{array} = 7 \times 10^3 + 4 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

$$10^3\ \ 10^2\ \ 10^1\ \ 10^0 = 7000 + 400 + 20 + 3 = 7423$$

The decimal number system has 10 digits, and the position values are integral powers of 10. We say that 10 is the *base* or *radix* of the decimal number system. Similarly, since the binary number system has two digits, its base is 2, and since the hex number system has 16 digits, its base is 16.

If 1101 is a binary number, the digits 1, 1, 0, and 1 represent 1×2^3, 1×2^2, 0×2^1, and 1×2^0, respectively:

binary numbers

$$\begin{array}{|c|c|c|c|}\hline 1 & 1 & 0 & 1 \\ \hline \end{array} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$2^3\ \ 2^2\ \ 2^1\ \ 2^0 = 8 + 4 + 0 + 1 = 13$$

If 7423 is a hex number, the digits 7, 4, 2, and 3 represent 7×16^3, 4×16^2, 2×16^1, and 3×16^0, respectively:

$$\begin{array}{|c|c|c|c|}\hline 7 & 4 & 3 & 2 \\ \hline \end{array} = 7 \times 16^3 + 4 \times 16^2 + 2 \times 16^1 + 3 \times 16^0$$

$$16^3\ \ 16^2\ \ 16^1\ \ 16^0 = 28672 + 1024 + 32 + 3 = 29731$$

F.2 Conversions between Binary and Decimal Numbers

binary to decimal

Given a binary number $b_n b_{n-1} b_{n-2} \ldots b_2 b_1 b_0$, the equivalent decimal value is

$$b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

Here are some examples of converting binary numbers to decimals:

Binary	*Conversion Formula*	*Decimal*
10	$1 \times 2^1 + 0 \times 2^0$	2
1000	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	8
10101011	$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	171

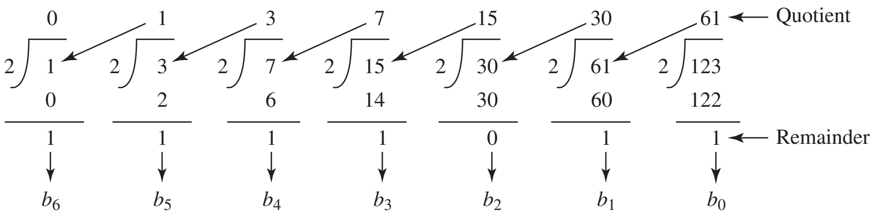
decimal to binary

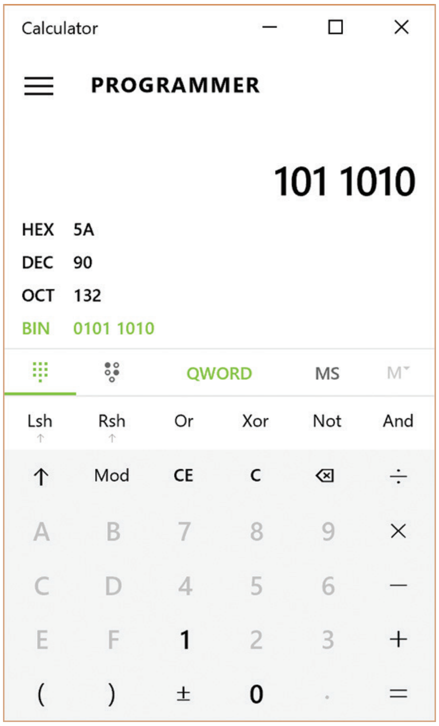
To convert a decimal number d to a binary number is to find the bits $b_n, b_{n-1}, b_{n-2}, \ldots, b_2, b$ and b_0 such that

$$d = b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

These bits can be found by successively dividing d by 2 until the quotient is 0. The remainders are $b_0, b_1, b_2, \ldots, b_{n-2}, b_{n-1}$, and b_n.

For example, the decimal number 123 is 1111011 in binary. The conversion is done as follows:


0
1
3
7
15
30
61
Quotient
2
1
2
3
2
7
2
15
2
30
2
61
2
123
0
2
6
14
30
60
122
1
1
1
1
0
1
1
Remainder
b_6
b_5
b_4
b_3
b_2
b_1
b_0


Calculator
PROGRAMMER

101 1010

HEX 5A

DEC 90

OCT 132

BIN 0101 1010

QWORD
MS
M˅
Lsh
Rsh
Or
Xor
Not
And
↑
Mod
CE
C
⌫
÷
A
B
7
8
9
×
C
D
4
5
6
−
E
F
1
2
3
+
(
)
±
0
.
=

FIGURE F.1 You can perform number conversions using the Windows Calculator.

Tip

The Windows Calculator, as shown in Figure F.1, is a useful tool for performing number conversions. To run it, search for *Calculator* from the *Start* button and launch Calculator, then under *View* select *Scientific*.

F.3 Conversions between Hexadecimal and Decimal Numbers

hex to decimal

Given a hexadecimal number $h_n h_{n-1} h_{n-2} \ldots h_2 h_1 h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

Here are some examples of converting hexadecimal numbers to decimals:

Hexadecimal	*Conversion Formula*	*Decimal*
7F	$7 \times 16^1 + 15 \times 16^0$	127
FFFF	$15 \times 16^3 + 15 \times 16^2 + 15 \times 16^1 + 15 \times 16^0$	65535
431	$4 \times 16^2 + 3 \times 16^1 + 1 \times 16^0$	1073

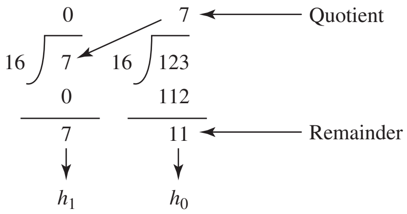
decimal to hex

To convert a decimal number d to a hexadecimal number is to find the hexadecimal digits $h_n, h_{n-1}, h_{n-2}, \ldots, h_2, h_1$, and h_0 such that

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

These numbers can be found by successively dividing d by 16 until the quotient is 0. The remainders are $h_0, h_1, h_2, \ldots, h_{n-2}, h_{n-1}$, and h_n.

For example, the decimal number 123 is 7B in hexadecimal. The conversion is done as follows:



F.4 Conversions between Binary and Hexadecimal Numbers

hex to binary

To convert a hexadecimal number to a binary number, simply convert each digit in the hexadecimal number into a four-digit binary number, using Table F.1.

For example, the hexadecimal number 7B is 01111011, where 7 is 0111 in binary and B is 1011 in binary.

binary to hex

To convert a binary number to a hexadecimal number, convert every four binary digits from right to left in the binary number into a hexadecimal number.

For example, the binary number 001110001101 is 38D, since 1101 is D, 1000 is 8, and 0011 is 3, as shown below.

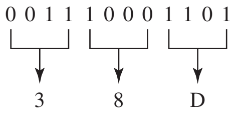


TABLE F.1 Converting Hexadecimal to Binary

Hexadecimal	*Binary*	*Decimal*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Note

Octal numbers are also useful. The octal number system has eight digits, 0 to 7. A decimal number 8 is represented in the octal system as 10.

Check Point

F.1 Convert the following decimal numbers into hexadecimal and binary numbers:

`100; 4340; 2000`

F.2 Convert the following binary numbers into hexadecimal and decimal numbers:

`1000011001; 100000000; 100111`

F.3 Convert the following hexadecimal numbers into binary and decimal numbers:

`FEFA9; 93; 2000`

APPENDIX G

Bitwise Operations

To write programs at the machine-level, often you need to deal with binary numbers directly and perform operations at the bit level. Java provides the bitwise operators and shift operators defined in Table G.1.

The bit operators apply only to integer types (`byte`, `short`, `int`, and `long`). A character involved in a bit operation is converted to an integer. All bitwise operators can form bitwise assignment operators, such as `^=`, `|=`, `<<=`, `>>=`, and `>>>=`.

TABLE G.1

Operator	*Name*	*Example (using bytes in the example)*	*Description*
`&`	Bitwise AND	10101110 `&` 10010010 yields 10000010	The AND of two corresponding bits yields a 1 if both bits are 1.
`|`	Bitwise inclusive OR	10101110 `|` 10010010 yields 10111110	The OR of two corresponding bits yields a 1 if either bit is 1.
`^`	Bitwise exclusive OR	10101110 `^` 10010010 yields 00111100	The XOR of two corresponding bits yields a 1 only if two bits are different.
`~`	One's complement	`~`10101110 yields 01010001	The operator toggles each bit from 0 to 1 and from 1 to 0.
`<<`	Left shift	10101110 `<<` 2 yields 10111000	The operator shifts bits in the first operand left by the number of bits specified in the second operand, filling with 0s on the right.
`>>`	Right shift with sign extension	10101110 `>>` 2 yields 11101011 00101110 `>>` 2 yields 00001011	The operator shifts bit in the first operand right by the number of bits specified in the second operand, filling with the highest (sign) bit on the left.
`>>>`	Unsigned right shift with zero extension	10101110 `>>>` 2 yields 00101011 00101110 `>>>` 2 yields 00001011	The operator shifts bit in the first operand right by the number of bits specified in the second operand, filling with 0s on the left.

Note

Programs using the bitwise operators are more efficient than the arithmetic operators. For example, to multiply an `int` value `x` by `2`, you can write `x << 1` rather than `x * 2`.

APPENDIX H

Regular Expressions

Often, you need to write the code to validate user input such as to check whether the input is a number, a string with all lowercase letters, or a Social Security number. How do you write this type of code? A simple and effective way to accomplish this task is to use the regular expression.

regular expression

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. Regular expression is a powerful tool for string manipulations. You can use regular expressions for matching, replacing, and splitting strings.

H.1 Matching Strings

matches

Let us begin with the `matches` method in the `String` class. At first glance, the `matches` method is very similar to the `equals` method. For example, the following two statements both evaluate to `true`:

```
"Java".matches("Java");
"Java".equals("Java");
```

However, the `matches` method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to `true`:

```
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

`"Java.*"` in the preceding statements is a regular expression. It describes a string pattern that begins with Java followed by any zero or more characters. Here, the substring `.*` matches any zero or more characters.

H.2 Regular Expression Syntax

A regular expression consists of literal characters and special symbols. Table H.1 lists some frequently used syntax for regular expressions.

TABLE H.1 Frequently Used Regular Expressions

Regular Expression	*Matches*	*Example*
`x`	`a specified character x`	`Java matches Java`
`.`	`any single character`	`Java` matches `J..a`
`(ab|cd)`	`ab or cd`	`ten matches t(en|im)`
`[abc]`	`a, b, or c`	`Java matches Ja[uvwx]a`

(*continued*)

Regular Expression	*Matches*	*Example*
`[^abc]`	any character except a, b, or c	`Java` matches `Ja[^ars]a`
`[a-z]`	a through z	`Java` matches `[A-M]av[a-d]`
`[^a-z]`	any character except a through z	`Java` matches `Jav[^b-d]`
`[a-e[m-p]]`	a through e or m through p	`Java` matches `[A-G[I-M]]av[a-d]`
`[a-e&&[c-p]]`	intersection of a-e with c-p	`Java` matches `[A-P&&[I-M]]av[a-d]`
`\d`	a digit, same as [0-9]	`Java2` matches `"Java[\\d]"`
`\D`	a non-digit	`$Java` matches `"[\\D][\\D]ava"`
`\w`	a word character	`Java1` matches `"[\\w]ava[\\d]"`
`\W`	a non-word character	`$Java` matches `"[\\W][\\w]ava"`
`\s`	a whitespace character	`"Java 2"` matches `"Java\\s2"`
`\S`	a non-whitespace char	`Java` matches `"[\\S]ava"`
*p**	zero or more occurrences of pattern *p*	`aaaa` matches `"a*"` `abab` matches `"(ab)*"`
p+	one or more occurrences of pattern *p*	`a` matches `"a+b*"` `able` matches `"(ab)+.*"`
p?	zero or one occurrence of pattern *p*	`Java` matches `"J?Java"` `ava` matches `"J?ava"`
p{n}	exactly n occurrences of pattern *p*	`Java` matches `"Ja{1}.*"` `Java` does not match `".{2}"`
p{n,}	at least n occurrences of pattern *p*	`aaaa` matches `"a{1,}"` `a` does not match `"a{2,}"`
p{n,m}	between n and m occurrences (inclusive)	`aaaa` matches `"a{1,9}"` `abb` does not match `"a{2,9}bb"`
`\p{P}`	a punctuation character !"#$%&'()*+, -./:;<=>?@ [\]^_‘{|}~	`J?a` matches `"J\\p{P}a"` `J?a.` does not match `"J\\p{P}a"`

Note

Backslash is a special character that starts an escape sequence in a string. So you need to use `\\` to represent a literal character `\`.

Note

Recall that a *whitespace character* is `' '`, `'\t'`, `'\n'`, `'\r'`, or `'\f'`. So `\s` is the same as `[ \t\n\r\f]`, and `\S` is the same as `[^ \t\n\r\f]`.

Note

A word character is any letter, digit, or the underscore character. So `\w` is the same as `[a-z[A-Z][0-9]_]` or simply `[a-zA-Z0-9_]`, and `\W` is the same as `[^a-zA-Z0-9_]`.

Note

The entries `*`, `+`, `?`, `{n}`, `{n,}`, and `{n, m}` in Table H.1 are called *quantifiers* that specify how many times the pattern before a quantifier may repeat. For example, `A*`

quantifier

matches zero or more `A`'s, `A+` matches one or more `A`'s, `A?` matches zero or one `A`, `A{3}` matches exactly `AAA`, `A{3,}` matches at least three `A`'s, and `A{3,6}` matches between 3 and 6 `A`'s. `*` is the same as `{0,}`, `+` is the same as `{1,}`, and `?` is the same as `{0,1}`.

Caution

Do not use spaces in the repeat quantifiers. For example, `A{3,6}` cannot be written as `A{3, 6}` with a space after the comma.

Note

You may use parentheses to group patterns. For example, `(ab){3}` matches `ababab`, but `ab{3}` matches `abbb`.

Let us use several examples to demonstrate how to construct regular expressions.

Example 1

The pattern for Social Security numbers is `xxx-xx-xxxx`, where `x` is a digit. A regular expression for Social Security numbers can be described as

```
[\\d]{3}-[\\d]{2}-[\\d]{4}
```

For example,

```
"111-22-3333".matches("[\\d]{3}-[\\d]{2}-[\\d]{4}") returns true.
"11-22-3333".matches("[\\d]{3}-[\\d]{2}-[\\d]{4}") returns false.
```

Example 2

An even number ends with digits `0`, `2`, `4`, `6`, or `8`. The pattern for even numbers can be described as

```
[\\d]*[02468]
```

For example,

```
"123".matches("[\\d]*[02468]") returns false.
"122".matches("[\\d]*[02468]") returns true.
```

Example 3

The pattern for telephone numbers is `(xxx) xxx-xxxx`, where `x` is a digit and the first digit cannot be zero. A regular expression for telephone numbers can be described as

```
\\([1-9][\\d]{2}\\) [\\d]{3}-[\\d]{4}
```

Note the parentheses symbols `(` and `)` are special characters in a regular expression for grouping patterns. To represent a literal `(` or `)` in a regular expression, you have to use `\\(` and `\\)`. For example,

```
"(912) 921-2728".matches("\\([1-9][\\d]{2}\\) [\\d]{3}-[\\d]{4}")
returns true.
"921-2728".matches("\\([1-9][\\d]{2}\\) [\\d]{3}-[\\d]{4}") returns
false.
```

Example 4

Suppose the last name consists of at most 25 letters, and the first letter is in uppercase. The pattern for a last name can be described as

```
[A-Z][a-zA-Z]{1,24}
```

Note you cannot have arbitrary whitespace in a regular expression. For example, `[A-Z][a-zA-Z]{1, 24}` would be wrong.
For example,

```
"Smith".matches("[A-Z][a-zA-Z]{1,24}") returns true.
"Jones123".matches("[A-Z][a-zA-Z]{1,24}") returns false.
```

Example 5

Java identifiers are defined in Section 2.3, Identifiers.

- An identifier must start with a letter, an underscore (_), or a dollar sign (`$`). It cannot start with a digit.
- An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs (`$`).

The pattern for identifiers can be described as

```
[a-zA-Z_$][\\w$]*
```

Example 6

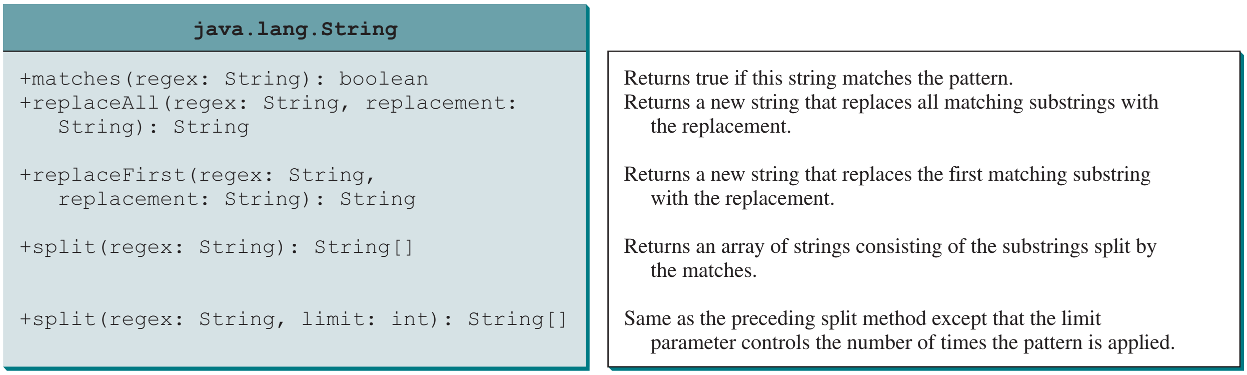
What strings are matched by the regular expression `"Welcome to (Java|HTML)"`? The answer is `Welcome to Java` or `Welcome to HTML`.

Example 7

What strings are matched by the regular expression `".*"`? The answer is any string.

H.3 Replacing and Splitting Strings

The `matches` method in the `String` class returns `true` if the string matches the regular expression. The `String` class also contains the `replaceAll`, `replaceFirst`, and `split` methods for replacing and splitting strings, as shown in Figure H.1.



java.lang.String	
`+matches(regex: String): boolean`	Returns true if this string matches the pattern.
`+replaceAll(regex: String, replacement: String): String`	Returns a new string that replaces all matching substrings with the replacement.
`+replaceFirst(regex: String, replacement: String): String`	Returns a new string that replaces the first matching substring with the replacement.
`+split(regex: String): String[]`	Returns an array of strings consisting of the substrings split by the matches.
`+split(regex: String, limit: int): String[]`	Same as the preceding split method except that the limit parameter controls the number of times the pattern is applied.

Figure H.1 The `String` class contains the methods for matching, replacing, and splitting strings using regular expressions.

The `replaceAll` method replaces all matching substring, and the `replaceFirst` method replaces the first matching substring. For example, the code

```
System.out.println("Java Java Java".replaceAll("v\\w", "wi"));
```

displays

```
Jawi Jawi Jawi
```

and this code

```
System.out.println("Java Java Java".replaceFirst("v\\w", "wi"));
```

displays

```
Jawi Java Java
```

There are two overloaded `split` methods. The `split(regex)` method splits a string into substrings delimited by the matches. For example, the statement

```
String[] tokens = "Java1HTML2Perl".split("\\d");
```

splits string `"Java1HTML2Perl"` into `Java`, `HTML`, and `Perl` and saves in `tokens[0]`, `tokens[1]`, and `tokens[2]`.

In the `split(regex, limit)` method, the `limit` parameter determines how many times the pattern is matched. If `limit <= 0`, `split(regex, limit)` is same as `split(regex)`. If `limit > 0`, the pattern is matched at most `limit – 1` times. Here are some examples:

```
"Java1HTML2Perl".split("\\d", 0); splits into Java, HTML, Perl
"Java1HTML2Perl".split("\\d", 1); splits into Java1HTML2Perl
"Java1HTML2Perl".split("\\d", 2); splits into Java, HTML2Perl
"Java1HTML2Perl".split("\\d", 3); splits into Java, HTML, Perl
"Java1HTML2Perl".split("\\d", 4); splits into Java, HTML, Perl
"Java1HTML2Perl".split("\\d", 5); splits into Java, HTML, Perl
```

Note

By default, all the quantifiers are *greedy*. This means that they will match as many occurrences as possible. For example, the following statement displays `JRvaa`, since the first match is `aaa`:

```
System.out.println("Jaaavaa".replaceFirst("a+", "R"));
```

You can change a qualifier's default behavior by appending a question mark (`?`) after it. The quantifier becomes *reluctant* or *lazy*, which means that it will match as few occurrences as possible. For example, the following statement displays `JRaavaa`, since the first match is `a`:

```
System.out.println("Jaaavaa".replaceFirst("a+?", "R"));
```

H.4 Replacing Partial Content in a Matched Substring

Sometimes, you need to make a replacement for partial content in a matched string. For example, suppose we have a text as follows:

```
String text = "3 * (x – y) is in lines 12-56.";
```

We would like to replace the text to

```
"3 * (x – y) is in lines 12 to 56.";
```

Note the `–` symbol is replaced to the word `"to"` if it is preceded by the word `"lines"` and between two numbers. We would like to replace all the occurrences of the `–` symbol by the word `"to"` for such cases in a text. To accomplish this, we will find a matching substring with the pattern `[lines \\d+-\\d+"]` and then replace the `–` symbol in the pattern by the word `"to"`. This can be done using the `Pattern` and `Matcher` classes.

The `Pattern` class is a compiled representation of a regular expression. You can create an instance of `Pattern` using `Pattern.compile(regex)`. The resulting pattern can then be used to create a `Matcher` object. For example, the following code creates a `Pattern` object `p` and creates a `Matcher` object `m` for the text using the pattern `p`:

```
String regex = "lines \\d+-\\d+";
Pattern p = Pattern.compile(regex);
Matcher m = p.matcher(text);
```

You can now use the `find()` method in the `Matcher` class to find a matched substring for the pattern, use the `group()` method to return the matched substring, and replace the `–` symbol in the matched string, and then use the `addReplacement` and `addTail` methods to add the text and its replacement into a `StringBuilder`.

The complete code is given in Listing H.1

LISTING H.1 PatternMatcherDemo.java

```
 1  import java.util.regex.Matcher;
 2  import java.util.regex.Pattern;
 3
 4  public class PatternMatcherDemo {
 5    public static void main(String args[]) {
 6     String text = "3 * (x – y) is in lines 12–56.";
 7      String regex = "lines \\d+-\\d+";
 8      Pattern p = Pattern.compile(regex);
 9      Matcher m = p.matcher(text);
10
11      StringBuffer sb = new StringBuffer();
12      while (m.find()) {
13         String replacement = m.group();
14         replacement = replacement.replace("–", " to ");
15         m.appendReplacement(sb, replacement);
16     }
17
18     m.appendTail(sb);
19     System.out.println(sb.toString());
20   }
26  }
```

This is an elaborated process. Invoking `m.find()` (line 12) scans the text to find the next match for the pattern in the text from the starting position. Initially, the starting position is at index `0`. Invoking `m.group()` (line 13) returns the matched substring to `String replacement`. The `String`'s `replace` method replaces `"-"` with `"to"` (line 14). Invoking `m.appendReplace(sb, replacement)` appends the current unmatched content in the text to `sb` and then appends `replacement` to `sb`, where `sb` is a `StringBuilder`. Note that the current unmatched content are the substring that has been scanned by `m.find()`, but not part of the matched string. The loop (lines 12-16) continues to find the next matching, obtain the matched substring, replace the partial content in the substring, and append the unmatched content and the replacement to `sb`. The loop ends when no more matches can be found. The program then invokes the `m.addTail(sb)` method to append the remaining unmatched content in the text to `sb` (line 18).

Note that all these methods `find()`, `group()`, `addReplacement`, and `addTail` are used together in a find-replace-append loop. When `m.find()` is invoked for the first time, the starting position is at index `0`. When `m.find()` is invoked again, it first resets the starting position to pass the end of the matched substring.

APPENDIX I

Enumerated Types

I.1 Simple Enumerated Types

An enumerated type defines a list of enumerated values. Each value is an identifier. For example, the following statement declares a type, named `MyFavoriteColor`, with values `RED`, `BLUE`, `GREEN`, and `YELLOW` in this order:

```
enum MyFavoriteColor {RED, BLUE, GREEN, YELLOW};
```

A value of an enumerated type is like a constant and so, by convention, is spelled with all uppercase letters. So, the preceding declaration uses `RED`, not `red`. By convention, an enumerated type is named like a class with first letter of each word capitalized.

Once a type is defined, you can declare a variable of that type:

```
MyFavoriteColor color;
```

The variable `color` can hold one of the values defined in the enumerated type `MyFavoriteColor` or `null`, but nothing else. Java enumerated type is *type-safe*, meaning that an attempt to assign a value other than one of the enumerated values or `null` will result in a compile error.

The enumerated values can be accessed using the syntax

```
EnumeratedTypeName.valueName
```

For example, the following statement assigns enumerated value `BLUE` to variable `color`:

```
color = MyFavoriteColor.BLUE;
```

Note you have to use the enumerated type name as a qualifier to reference a value such as `BLUE`.

As with any other type, you can declare and initialize a variable in one statement:

```
MyFavoriteColor color = MyFavoriteColor.BLUE;
```

An enumerated type is treated as a special class. An enumerated type variable is therefore a reference variable. An enumerated type is a subtype of the `Object` class and the `Comparable` interface. Therefore, an enumerated type inherits all the methods in the `Object` class and the `compareTo` method in the `Comparable` interface. Additionally, you can use the following methods on an enumerated object:

- `public String name();`
  Returns a name of the value for the object.
- `public int ordinal();`
  Returns the ordinal value associated with the enumerated value. The first value in an enumerated type has an ordinal value of 0, the second has an ordinal value of 1, the third one 3, and so on.

Listing I.1 gives a program that demonstrates the use of enumerated types.

LISTING I.1 `EnumeratedTypeDemo.java`

```
 1  public class EnumeratedTypeDemo {
 2    static enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
 3      FRIDAY, SATURDAY};
 4
 5    public static void main(String[] args) {
 6      Day day1 = Day.FRIDAY;
 7      Day day2 = Day.THURSDAY;
 8
 9      System.out.println("day1's name is " + day1.name());
10      System.out.println("day2's name is " + day2.name());
11      System.out.println("day1's ordinal is " + day1.ordinal());
12      System.out.println("day2's ordinal is " + day2.ordinal());
13
14      System.out.println("day1.equals(day2) returns " +
15        day1.equals(day2));
16      System.out.println("day1.toString() returns " +
17        day1.toString());
18      System.out.println("day1.compareTo(day2) returns " +
19        day1.compareTo(day2));
20    }
21  }
```

define an enum type

declare an enum variable

get enum name

get enum ordinal

compare enum values

```
day1's name is FRIDAY
day2's name is THURSDAY
day1's ordinal is 5
day2's ordinal is 4
day1.equals(day2) returns false
day1.toString() returns FRIDAY
day1.compareTo(day2) returns 1
```

An enumerated type `Day` is defined in lines 2 and 3. Variables `day1` and `day2` are declared as the `Day` type and assigned enumerated values in lines 6 and 7. Since `day1`'s value is `FRIDAY`, its ordinal value is 5 (line 11). Since `day2`'s value is `THURSDAY`, its ordinal value is 4 (line 12).

Since an enumerated type is a subclass of the `Object` class and the `Comparable` interface, you can invoke the methods `equals`, `toString`, and `compareTo` from an enumerated object reference variable (lines 14–19). `day1.equals(day2)` returns true if `day1` and `day2` have the same ordinal value. `day1.compareTo(day2)` returns the difference between `day1`'s ordinal value and `day2`'s.

Alternatively, you can rewrite the code in Listing I.1 into Listing I.2.

LISTING I.2 `StandaloneEnumTypeDemo.java`

```
public class StandaloneEnumTypeDemo {
  public static void main(String[] args) {
    Day day1 = Day.FRIDAY;
    Day day2 = Day.THURSDAY;

    System.out.println("day1's name is " + day1.name());
    System.out.println("day2's name is " + day2.name());
    System.out.println("day1's ordinal is " + day1.ordinal());
```

```
 9       System.out.println("day2's ordinal is " + day2.ordinal());
10
11       System.out.println("day1.equals(day2) returns " +
12         day1.equals(day2));
13       System.out.println("day1.toString() returns " +
14         day1.toString());
15       System.out.println("day1.compareTo(day2) returns " +
16         day1.compareTo(day2));
17     }
18   }
19
20   enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
21     FRIDAY, SATURDAY}
```

An enumerated type can be defined inside a class, as shown in lines 2 and 3 in Listing I.1, or standalone as shown in lines 20 and 21 in Listing I.2. In the former case, the type is treated as an inner class. After the program is compiled, a class named EnumeratedTypeDemo$Day.class is created. In the latter case, the type is treated as a stand-alone class. After the program is compiled, a class named Day.class is created.

Note

When an enumerated type is declared inside a class, the type must be declared as a member of the class and cannot be declared inside a method. Furthermore, the type is always `static`. For this reason, the `static` keyword in line 2 in Listing I.1 may be omitted. The visibility modifiers on inner class can be also be applied to enumerated types defined inside a class.

Tip

Using enumerated values (e.g., `Day.MONDAY`, `Day.TUESDAY`, and so on) rather than literal integer values (e.g., 0, 1, and so on) can make the program easier to read and maintain.

I.2 Using `if` or `switch` Statements with an Enumerated Variable

An enumerated variable holds a value. Often, your program needs to perform a specific action depending on the value. For example, if the value is `Day.MONDAY`, play soccer; if the value is `Day.TUESDAY`, take piano lesson, and so on. You can use an `if` statement or a `switch` statement to test the value in the variable, as shown in (a) and (b).

```
if (day.equals(Day.MONDAY)) {
  // process Monday
}
else if (day.equals(Day.TUESDAY)) {
  // process Tuesday
}
else
  ...
```

(a)

Equivalent

```
switch (day) {
  case MONDAY:
    // process Monday
    break;
  case TUESDAY:
    // process Tuesday
    break;
  ...
}
```

(b)

In the `switch` statement in (b), the case label is an unqualified enumerated value (e.g., `MONDAY`, but not `Day.MONDAY`).

I.3 Processing Enumerated Values Using a Foreach Loop

Each enumerated type has a static method `values()` that returns all enumerated values for the type in an array. For example,

```
Day[] days = Day.values();
```

You can use a regular for loop in (a) or foreach loop in (b) to process all the values in the array.

```
for (int i = 0; i < days.length; i++)
  System.out.println(days[i]);
```

(a)

Equivalent

```
for (Day day: days)
  System.out.println(day);
```

(b)

I.4 Enumerated Types with Data Fields, Constructors, and Methods

The simple enumerated types introduced in the preceding section define a type with a list of enumerated values. You can also define an enumerate type with data fields, constructors, and methods, as shown in Listing I.3.

Listing I.3 TrafficLight.java

```
 1  public enum TrafficLight {
 2    RED ("Please stop"), GREEN ("Please go"),
 3    YELLOW ("Please caution");
 4
 5    private String description;
 6
 7    private TrafficLight(String description) {
 8      this.description = description;
 9    }
10
11    public String getDescription() {
12      return description;
13    }
14  }
```

The enumerated values are defined in lines 2 and 3. The value declaration must be the first statement in the type declaration. A data field named `description` is declared in line 5 to describe an enumerated value. The constructor `TrafficLight` is declared in lines 7–9. The constructor is invoked whenever an enumerated value is accessed. The enumerated value's argument is passed to the constructor, which is then assigned to `description`.

Listing I.4 gives a test program to use `TrafficLight`.

Listing I.4 TestTrafficLight.java

```
public class TestTrafficLight {
  public static void main(String[] args) {
    TrafficLight light = TrafficLight.RED;
    System.out.println(light.getDescription());
  }
}
```

An enumerated value `TrafficLight.red` is assigned to variable `light` (line 3). Accessing `TrafficLight.RED` causes the JVM to invoke the constructor with argument “please stop”. The methods in enumerated type are invoked in the same way as the methods in a class. `light.getDescription()` returns the description for the enumerated value (line 4).

Note

The Java syntax requires that the constructor for enumerated types be private to prevent it from being invoked directly. The private modifier may be omitted. In this case, it is considered private by default.

APPENDIX J

The Big-O, Big-Omega, and Big-Theta Notations

Chapter 22 presented the Big-O notation in laymen's term. In this appendix, we give a precise mathematic definition for the Big-O notation. We will also present the Big-Omega and Big-Theta notations.

J.1 The Big-O Notation

The Big-O notation is an asymptotic notation that describes the behavior of a function when its argument approaches a particular value or infinity. Let $f(n)$ and $g(n)$ be two functions, we say that $f(n)$ is $O(g(n))$, pronounced "big-O of $g(n)$", if there is a constant c $(c > 0)$ and value m such that $f(n) \leq c \times g(n)$, for $n \geq m$.

For example, $f(n) = 5n^3 + 8n^2$ is $O(n^3)$, because you can find $c = 13$ and $m = 1$ such that $f(n) \leq cn^3$ for $n \geq m$. $f(n) = 6n\log n + n^2$ is $O(n^2)$, because you can find $c = 7$ and $m = 2$ such that $f(n) \leq cn^2$ for $n \geq m$. $f(n) = 6n\log n + 400n$ is $O(n\log n)$, because you can find $c = 406$ and $m = 2$ such that $f(n) \leq cn\log n$ for $n \geq m$. $f(n) = n^2$ is $O(n^3)$, because you can find $c = 1$ and $m = 1$ such that $f(n) \leq cn^3$ for $n \geq m$. Note that there are infinite number of choices of c and m such that $f(n) \leq c \times g(n)$ for $n \geq m$.

The Big-O notation denotes that a function $f(n)$ is asymptotically less than or equal to another function $g(n)$. This allows you to simplify the function by ignoring multiplicative constants and discarding the non-dominating terms in the function.

J.2 The Big-Omega Notation

The Big-Omega notation is the opposite of the Big-O notation. It is an asymptotic notation that denotes that a function $f(n)$ is greater than or equal to another function $g(n)$. Let $f(n)$ and $g(n)$ be two functions, we say that $f(n)$ is $\Omega(g(n))$, pronounced "big-Omega of $g(n)$", if there is a constant $c(c > 0)$ and value m such that $f(n) \geq c \times g(n)$, for $n \geq m$.

For example, $f(n) = 5\mathrm{n}^3 + 8\mathrm{n}^2$ is $\Omega(\mathrm{n}^3)$, because you can find $c = 5$ and $m = 1$ such that $f(n) \geq cn^3$ for $n \geq m$. $f(n) = 6n\log n + n^2$ is $\Omega(n^2)$, because you can find $c = 1$ and $m = 1$ such that $f(n) \geq cn^2$ for $n \geq m$. $f(n) = 6n\log n + 400n$ is $\Omega(n\log n)$, because you can find $c = 6$ and $m = 1$ such that $f(n) \geq cn\log n$ for $n \geq m$. $f(n) = n^2$ is $\Omega(n)$, because you can find $c = 1$ and $m = 1$ such that $f(n) \geq cn$ for $n \geq m$. Note that there are infinite number of choices of c and m such that $f(n) \geq c \times g(n)$ for $n \geq m$.

J.3 The Big-Theta Notation

The Big-Theta notation denotes that two functions are the same asymptotically. Let $f(n)$ and $g(n)$ be two functions, we say that $f(n)$ is $\Theta(g(n))$, pronounced "big-Theta of $g(n)$", if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

For example, $f(n) = 5n^3 + 8n^2$ is $\Theta(n^3)$, because you $f(n)$ is $O(n^3)$ and $f(n)$ is $\Omega(n^3)$. $f(n) = 6n\log n + 400n$ is $\Theta(n\log n)$, because $f(n)$ is $O(n\log n)$ and $f(n)$ is $\Theta(n\log n)$.

Note

The Big-O notation gives an upper bound of a function. The Big-Omega notation gives a lower bound of a function. The Big-Theta notation gives a tight bound of a function. For simplicity, the Big-O notation is often used, even though the Big-Theta notation may be more factually appropriate.

Java Quick Reference

Console Input

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
long longValue = input.nextLong();
double doubleValue = input.nextDouble();
float floatValue = input.nextFloat();
String string = input.next();
String line = input.nextLine();
```

Console Output

```
System.out.println(anyValue);
```

Conditional Expression

```
boolean-expression ? expression1 :
  expression2
```

```
y = (x > 0) ? 1 : -1
```

```
System.out.println(number % 2 == 0 ?
  "number is even" : "number is odd");
```

Primitive Data Types

byte	8 bits
short	16 bits
int	32 bits
long	64 bits
float	32 bits
double	64 bits
char	16 bits
boolean	true/false

Arithmetic Operators

+	addition
-	subtraction
*****	multiplication
/	division
%	remainder
++var	preincrement
--var	predecrement
var++	postincrement
var--	postdecrement

Assignment Operators

=	assignment
+=	addition assignment
-=	subtraction assignment
***=**	multiplication assignment
/=	division assignment
%=	remainder assignment

Relational Operators

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal

Logical Operators

&&	short circuit AND
||	short circuit OR
!	NOT
^	exclusive OR

if Statements

```
if (condition) {
  statements;
}
```

```
if (condition) {
  statements;
}
else {
  statements;
}
```

```
if (condition1) {
  statements;
}
else if (condition2) {
  statements;
}
else {
  statements;
}
```

switch Statements

```
switch (intExpression) {
  case value1:
    statements;
    break;
  ...
  case valuen:
    statements;
    break;
  default:
    statements;
}
```

loop Statements

```
while (condition) {
  statements;
}
```

```
do {
  statements;
} while (condition);
```

```
for (init; condition;
  adjustment) {
  statements;
}
```

Companion Website: www.pearsonhighered.com/liang

Java Quick Reference

Frequently Used Static Constants/Methods

```
Math.PI
Math.random()
Math.pow(a, b)
Math.abs(a)
Math.max(a, b)
Math.min(a, b)
Math.sqrt(a)
Math.sin(radians)
Math.asin(a)
Math.toRadians(degrees)
Math.toDegress(radians)
System.currentTimeMillis()
Integer.parseInt(string)
Integer.parseInt(string, radix)
Double.parseDouble(string)
Arrays.sort(type[] list)
Arrays.binarySearch(type[] list, type key)
```

Array/Length/Initializer

```
 int[] list = new int[10];
 list.length;
 int[] list = {1, 2, 3, 4};
```

Multidimensional Array/Length/Initializer

```
 int[][] list = new int[10][10];
 list.length;
 list[0].length;
 int[][] list = {{1, 2}, {3, 4}};
```

Ragged Array

```
 int[][] m = {{1, 2, 3, 4},
             {1, 2, 3},
             {1, 2},
             {1}};
```

Text File Output

```
 PrintWriter output =
   new PrintWriter(filename);
 output.print(...);
 output.println(...);
 output.printf(...);
```

Text File Input

```
  Scanner input = new Scanner(
   new File(filename));
```

File Class

```
File file =
  new File(filename);
file.exists()
file.renameTo(File)
file.delete()
```

Object Class

```
Object o = new Object();
o.toString();
o.equals(o1);
```

Comparable Interface

```
c.compareTo(Comparable)
c is a Comparable object
```

String Class

```
String s = "Welcome";
String s = new String(char[]);
int length = s.length();
char ch = s.charAt(index);
int d = s.compareTo(s1);
boolean b = s.equals(s1);
boolean b = s.startsWith(s1);
boolean b = s.endsWith(s1);
boolean b = s.contains(s1);
String s1 = s.trim();
String s1 = s.toUpperCase();
String s1 = s.toLowerCase();
int index = s.indexOf(ch);
int index = s.lastIndexOf(ch);
String s1 = s.substring(ch);
String s1 = s.substring(i,j);
char[] chs = s.toCharArray();
boolean b = s.matches(regex);
String s1 = s.replaceAll(regex,repl);
String[] tokens = s.split(regex);
```

ArrayList Class

```
ArrayList<E> list = new ArrayList<>();
list.add(object);
list.add(index, object);
list.clear();
Object o = list.get(index);
boolean b = list.isEmpty();
boolean b = list.contains(object);
int i = list.size();
list.remove(index);
list.set(index, object);
int i = list.indexOf(object);
int i = list.lastIndexOf(object);
```

printf Method

```
System.out.printf("%b %c %d %f %e %s",
  true, 'A', 45, 45.5, 45.5, "Welcome");
System.out.printf("%-5d %10.2f %10.2e %8s",
  45, 45.5, 45.5, "Welcome");
```

Companion Website: www.pearsonhighered.com/liang

INDEX

Symbols

`--` (decrement operator), 57–58
`-` (subtraction operator), 46, 53
`.` (dot operator), 333
`.` (object member access operator), 333, 431
`/` (division operator), 46, 53
`//`, in line comment syntax, 18
`/*`, in block comment syntax, 18
`/**.*/` (Javadoc comment syntax), 18
`/=` (division assignment operator), 56–57
`;` (semicolons), common errors, 86
`\` (backslash character), as directory separator, 478
`\` (escape characters), 128
`||` (or logical operator), 95–99
`+` (addition operator), 46, 53
`+` (string concatenation operator), 36, 133
`++` (increment operator), 57–58
`+=` (addition assignment operator), augmented, 56–57
`=` (assignment operator), 42–43, 56–57
`=` (equals operator), 78
`-=` (subtraction assignment operator), 56–57
`==` (comparison operator), 78, 434
`=` (equal to operator), 78
`!` (not logical operator), 95–99
`!=` (not equal to comparison operator), 78
`$` (dollar sign character), use in source code, 40
`%` (remainder or modulo operator), 46, 53
`%=` (remainder assignment operator), 56–57
`&&` (and logical operator), 95–99
`( )` (parentheses), 13, 227
`*` (multiplication operator), 14, 46, 53
`*=` (multiplication assignment operator), 56
`^` (exclusive or logical operator), 95–99
`{}` (curly braces), 13, 81, 85
`<` (less than comparison operator), 78
`<=` (less than or equal to comparison operator), 78
`>` (greater than comparison operator), 78
`>=` (greater than or equal to comparison operator), 78

Numbers

24-point game, 811–812

A

`abs` method, `Math` class, 124, 530
Absolute file name, 477
Abstract classes
  `AbstractCollection` class, 776, 777
  `AbstractMap` class, 830
  `AbstractSet` class, 816
  case study: abstract number class, 505–507
  case study: `Calendar` and `GregorianCalendar` classes, 507–510
  characteristics of, 504–505
  `Circle.java` and `Rectangle.java` examples, 502
  compared with interfaces, 523–526
  `GeometricObject.java` example, 500–502
  `InputStream` and `OutputStream` classes, 694–695
  interfaces compared to, 510
  key terms, 534
  overview of, 368–369, 500–501
  questions and excercises, 535–540
  `Rational.java` example, 528–531
  reasons for using abstract methods, 502
  summary, 534–535
  `TestCalendar.java` example, 508–510
  `TestGeometricObject.java` example, 502–503
  `TestRationalClass.java` example, 527–528
  using as interface, 924
Abstract data type (ADT), 368
Abstract methods
  characteristics of, 504
  `GenericMatrix.java` example, 766–769
  `GeometricObject` class, 501–502
  implementing in subclasses, 501–502
  in interfaces, 510
  key terms, 534
  in `Number` class, 530
  overview of, 227–228
  questions and exercises, 535–540
  reasons for using, 502
  summary, 534–535
`abstract` modifier, for denoting abstract methods, 500
Abstract number class
  `LargestNumbers.java`, 506–507
  overview of, 505–507
Abstract Windows Toolkit. *see* AWT (Abstract Windows Toolkit)
`AbstractCollection` class, 777
`AbstractMap` class, 830
`AbstractSet` class, 816
Accessibility modifiers, 1169
Accessor methods. *see* Getter (accessor) methods
`acos` method, trigonometry, 122–123
`ActionEvent`, 595–596
Actions (behaviors), object, 324
Activation records, invoking methods and, 210
Actual concrete types, 752
Actual parameters, defining methods and, 207
Ada, high-level languages, 8
`add` method
  implementing linked lists, 935
  `List` interface, 783
Addition (`+=`) assignment operator, 56–57
Addition (`+`) operator, 46, 53
Adelson-Velsky, G. M., 996
Adjacency lists, representing edges, 1052–1054
Adjacency matrices
  representing edges, 1052–1054
  weighted, 1054
Adjacent edges, overview of, 1048
ADT (Abstract data type), 368
Aggregate operations, for collection streams
  `AnalyzeNumbersUsingStream.java` example, 1150–1151
  case study: analyzing numbers, 1150–1151
  case study: counting keywords, 1155–1156
  case study: counting occurrences of each letter, 1151–1152
  case study: counting occurrences of each letter in string, 1152–1153
  case study: finding directory size, 1154–1155
  case study: occurrences of words, 1157–1158
  case study: processing all elements in two-dimensional array, 1153–1154

Aggregate operations, for collection streams (*Continued*)
  `CollectDemo.java` example, 1145–1147
  `CollectGroupDemo.java` example, 1148–1150
  `CountKeywordStream.java` example, 1155–1156
  `CountLettersUsingStream.java` example, 1151–1152
  `CountOccurrenceOfLettersInAString.java` example, 1152–1153
  `CountOccurrenceOfWordsStream.java` example, 1157–1158
  `DirectorySizeStream.java` example, 1154–1155
  `DoubleStream`, 1136–1139
  grouping elements using `groupingby` collector, 1147–1150
  `IntStream`, 1136–1139
  `IntStreamDemo.java` example, 1136–1139
  `LongStream`, 1136–1139
  overview of, 1130
  parallel streams, 1139–1141
  `ParallelStreamDemo.java` example, 1139–1141
  quiz and exercises, 1158–1159
  `Stream` class, 1131
  stream pipelines, 1130–1136
  stream reduction using `collect` method, 1144–1147
  stream reduction using `reduce` method, 1141–1144
  `StreamDemo.java` example, 1132–1133
  `StreamReductionDemo.java` example, 1142–1144
  summary, 1158
  `TwoDimensionalArrayStream.java` example, 1153–1154
Aggregating classes, 376
Aggregating objects, 376
Aggregation relationships, objects, 376–377
AIFF audio files, 676
Algorithms, 34
  analyzing Towers of Hanoi problem, 846–847
  Big *O* notation for measuring efficiency of, 840–841
  binary search, 846
  Boyer-Moore algorithm, 870–873
  bubble sort, 890–892
  comparing growth functions, 847–848
  comparing prime numbers, 861
  determining Big *O* for repetition, sequence, and selection statements, 842–845
  `EfficientPrimeNumbers.java` example, 857–859
  external sorts. *see* External sorts
  finding closest pair of points, 861–864
  finding convex hull for a set of points, 867–869
  finding Fibonacci numbers, 849–851
  finding greatest common denominator, 851–855
  finding prime numbers, 855–861
  `GCDEuclid.java` example, 853–855
  `GCD.java` example, 852–853
  gift-wrapping algorithm, 867–868
  Graham's algorithm, 868–869
  graph algorithms, 1046–1047
  greedy, 986
  heap sort. *see* Heap sorts
  insertion sorts, 888–890
  key terms, 876
  Knuth-Morris-Pratt algorithm, 873–876
  merge sorts, 892–896
  overview of, 840
  `PrimeNumbers.java` example, 856–857
  quick sort, 896–900
  quiz and exercises, 878–886
  recurrence relations and, 847
  selection sort and insertion sort, 846
  `SieveOfEratosthenes.java` example, 860–861
  solving Eight Queens problem, 864–867
  for sort method, 759
  summary, 876–877
Algorithms, spanning tree
  Dijkstra's single-source shortest-path algorithm, 1106–1111
  MST algorithm, 1103–1105
  Prim's minimum spanning tree algorithm, 1103–1105
`allMatch` method, 1132, 1134
Ambiguous invocation, of methods, 223
American Standard Code for Information Interchange (ASCII). *see* ASCII (American Standard Code for Information Interchange)
And (`&&`) logical operator, 95–99
Animation
  case study: bouncing ball, 626–629
  case study: US map, 630–633
  `ClockAnimation.java`, 625–626
  `FadeTransition`, 622–623
  key terms, 633
  `PathTransition`, 619–622
  programming exercises, 634–641
  quiz, 634
  summary, 633–634
  `Timeline`, 624–625
Anonymous arrays, 262
Anonymous objects, 333
`AnonymousHandlerDemo.java`, 603–605
`anyMatch` method, 1132, 1134
Application Program Interfaces (APIs), 11
Apps, developing on Web servers, 11
`Arc`
  overview, 575
  `ShowArc.java`, 575–577
Arguments
  defining methods and, 207
  passing by values, 214–217
  receiving string arguments from command line, 276–279
  variable-length argument lists, 268–269
`ArithmeticException` class, 457
Arithmetic/logic units, CPU components, 3
Array elements, 252
Array initializers, 252–253
`arraycopy` method, `System` class, 260
`ArrayIndexOutOfBoundsException`, 255
`ArrayList` class
  animation of array lists, 925
  case study: custom stack class, 441–442
  cloning arrays, 519
  compared with `LinkedList`, 784–787
  creating and adding numbers to array lists, 434–440
  creating array lists, 778–780
  defined under `List` interface, 783
  `DistinctNumbers.java` example, 438–440
  as example of generic class, 752–753
  heap sorts, 901
  implementing array lists, 928–935
  implementing bucket sorts, 908
  implementing buckets, 1023
  implementing stacks using array lists. *see* Stacks
  `MyArrayList`, 924–925, 947
  `MyArrayList` compared with `MyLinkedList`, 947
  `MyArrayList.java` example, 929–933
  `MyList.java` example, 926–928
  representing edges in graphs, 1053
  `SetListPerformanceTest.java` example, 825–826
  storing edge objects in, 1051
  for storing elements in a list, 776
  storing heaps in, 901
  storing list of objects in, 434–435
  `TestArrayAndLinkedList.java`, 785–787
  `TestArrayList.java` example, 436–438

  `TestMyArrayList.java` example, 933–935
  `Vector` class compared with, 799
`Arrays` class, 274–276
Arrays, in general
  edge arrays, 1051
  as fixed-size data structure, 928
  implementing binary heaps using, 901
  ragged arrays, 1052
  sorting using `Heap` class, 906
  storing lists in. *see* `ArrayList` class
  storing vertices in, 1050
Arrays, multi-dimensional
  case study: daily temperature and humidity, 304–306
  case study: guessing birthdays, 306–307
  overview of, 303–304
  questions and exercises, 308–321
  summary, 307
Arrays, single-dimensional
  accessing elements, 252
  `ArrayList` class, 437–438
  `Arrays` class, 274–276
  case study: analyzing numbers, 257–258
  case study: counting occurrences of letters, 265–268
  case study: deck of cards, 258–260
  case study: generic method for sorting, 758–759
  constructing strings from, 388
  converting strings to/from, 392
  copying, 260–261
  creating, 251–252, 514–516
  declaring, 250
  foreach loops, 255–257
  initializers, 252–253
  key terms, 279
  of objects, 353–355
  overview of, 250
  passing to methods, 261–264
  processing, 253–255
  questions and exercises, 280–288
  returning from methods, 264–265
  searching, 269–273
  serializing, 709–711
  size and default values, 252
  sorting, 273–274, 514–516
  summary, 279–280
  treating as objects in Java, 333
  variable-length argument lists, 268–269
Arrays, two-dimensional
  case study: finding closest pair of points, 298–300
  case study: grading multiple-choice test, 296–298
  case study: processing all elements in a two-dimensional array, 1153–1154
  case study: Sudoku, 300–303
  declaring variables and creating two-dimensional arrays, 290–291
  defined, 290
  obtaining length of two-dimensional arrays, 291–292
  passing to methods to two-dimensional arrays, 295–296
  processing two-dimensional arrays, 293–295
  questions and exercises, 308–321
  ragged arrays, 292–293
  representing graph edges with, 1051
  representing weighted graphs, 1093–1095
  summary, 307
Arrows keys, on keyboards, 6
ASCII (American Standard Code for Information Interchange)
  character (`char` data type), 126–127
  decimal and hexadecimal equivalents, 1165
  encoding scheme, 3–4
  text encoding, 692
  text I/O *vs.* binary I/O, 693–694
`asin` method, trigonometry, 123
`asList` method, 786
Assemblers, 7
Assembly language, 7
Assignment operator (`=`), 1175
  augmented, 56–57
  overview of, 42–43
Assignment statements (assignment expressions)
  assigning value to variables, 36
  overview of, 42–43
Associative array, 1016
Associative arrays. *see* Maps
Associativity, of operators, 107, 1166–1167
`atan` method, trigonometry, 122–123
Attributes, object, 324
Audio files
  case study: national flags and anthems, 679–681
  `MediaDemo.java`, 677–679
Autoboxing/Autounboxing, 386, 753–754
Average-case analysis
  measuring algorithm efficiency, 840, 854
  quick sort and, 899–900
AVL trees
  `AVLTree.java` example, 1002–1007
  balancing nodes on a path, 1000–1001
  deleting elements, 1002
  designing classes for, 999–1000
  key terms, 1012
  overriding the `insert` method, 1000–1001
  overview of, 996
  questions and exercises, 1012–1013
  rebalancing, 996–998
  rotations for balancing, 1001
  summary, 1012
  `TestAVLTree.java` example, 1008–1011
  time complexity of, 1011
`AVLTree` class
  `delete` method, 1007, 1011
  overview of, 1002–1007
  as subclass of `BST` class, 999
  testing, 1008–1011
AWT (Abstract Windows Toolkit)
  `Color` class, 553–554
  `Date` class, 336–337, 507–508
  `Error` class, 460, 462
  event classes in, 596
  `EventObject` class, 596–597
  exceptions. *see* `Exception` class
  `File` class, 477–479, 692
  `Font` class, 554–555
  `GeometricObject` class, 501–502
  `GuessDate` class, 306–307
  `IllegalArgumentException` class, 463
  `InputMismatchException` class, 458, 484
  `KeyEvent` class, 613
  `MalformedURLException` class, 487–488
  `MouseEvent` class, 611–612
  `Polygon` class, 577–578
  `String` class, 388
  Swing *vs.*, 542

B

Babylonian method, 242
Backslash character (`\`), as directory separator, 478
Backtracking algorithm, 864–867

Backward pointer, in doubly linked lists, 948
Balance factor, for AVL nodes, 996, 1004
Balanced nodes
  in AVL trees, 996
  `AVLTree` class, 1002–1003, 1006–1007
Base cases, in recursion, 726
`BaseStream` interface, 1130
BASIC, high-level languages, 8
Bean machine game, 288, 640
`beginIndex` method, for obtaining substrings from strings, 137
Behaviors (actions), object, 324
Behind the scene evaluation, expressions, 107
Best-case input
  measuring algorithm efficiency, 840, 854
  quick sort and, 899–900
BFS (breadth-first searches). *see* Breadth-first searches (BFS)
Big *O*
  determining for repetition, sequence, and selection statements, 842–845
  for measuring algorithm efficiency, 840–842
`BigDecimal` class, 387–388, 505
Binary
  files, 692
  machine language as binary code, 7
  operator, 48
  searches, 270–273, 730
Binary digits (Bits), 3
Binary heaps (binary trees), 900. *see also* Heap sorts
  complete, 900, 906
Binary I/O
  `BufferedInputStream` and `BufferedOutputStream` classes, 701–704
  characters and strings in, 698–699
  classes, 694–704
  `DataInputStream` and `DataOutputStream` classes, 698–701
  `DetectEndOfFile.java`, 701
  `FileInputStream` and `FileOutputStream` classes, 695–698
  `FilterInputStream` and `FilterOutputStream` classes, 698
  overview of, 692
  `TestDataStream.java`, 699–700
  `TestFileStream.java`, 696–698
  vs. text I/O, 693–694
Binary numbers
  converting to/from decimal, 745, 1172
  converting to/from hexadecimal, 1173–1174
  overview of, 1171
Binary search algorithm, 883
  analyzing, 846
  recurrence relations and, 847
Binary search trees (BST)
  `BST` class, 965–974
  `BSTAnimation.java` example, 980–981
  `BST.java` example, 968–973
  `BTView.java` example, 981–983
  case study: data compression, 985–990
  deleting elements, 974–980
  displaying/visualizing binary trees, 980–983
  `HuffmanCode.java` example, 987–990
  implementing using linked structure, 960–961
  inserting elements, 962–963
  iterators, 983–985
  key terms, 990
  overview of, 960
  quiz and exercises, 990–994
  representation of, 961–962
  searching for elements, 962
  summary, 990
  `TestBSTDelete.java` example, 977–980
  `TestBST.java` example, 973–974
  `TestBSTWithIterator.java` example, 984–985
  tree traversal, 963–964
  tree visualization and MVC, 980–983
  `Tree.java` example, 965–967
Binary trees, 960
`binarySearch` method
  applying to lists, 792–793
  `Arrays` class, 275
Binding properties
  `BindingDemo.java`, 550–551
  `ShowCircleCentered.java`, 548–550
Bit operators, 1175
Bits (binary digits), 3
Bitwise operators, 1175
Block comments, in `Welcome.java`, 13
Block modifiers, 1168–1169
Block style, programming style, 19
Blocks, in `Welcome.java`, 13
BMI (Body Mass Index), 91–92, 372–375
Boolean accessor method, 347
`boolean` data type
  `java.util.Random`, 337
  overview of, 78–80
Boolean expressions
  case study: determining leap year, 99–100
  conditional operators, 105–106
  defined, 78
  `if` statements and, 80–81
  `if-else` statements, 82–83
  writing, 88–89
Boolean literals, 79
Boolean values
  defined, 78
  as format specifier, 147
  logical operators and, 95
  redundancy in testing, 86
Boolean variables
  assigning, 88
  overview of, 78–79
  redundancy in testing, 86
`BorderPane`
  overview of, 563
  `ShowBorderPane.java`, 563–564
Bottom-up implementation, 229–231
Bounded generic types
  erasing, 764–765
  `GenericMatrix.java` example, 767–769
  `MaxUsingGenericType.java` example, 760–761
  overview of, 757
Bounded wildcards, 762
Boxing, converting wrapper object to primitive value, 386
Boyer-Moore algorithm, 870–873
  `StringMatchBoyerMoore.java`, 872
Braces. *see* Curly braces (`{}`)
Breadth-first searches (BFS)
  applications of, 1077
  finding BFS trees, 1048
  implementing, 1075–1076
  overview of, 1074
  `TestBFS.java`, 1076
  traversing graphs, 1067
Breadth-first traversal, tree traversal, 964
`break` statements
  controlling loops, 186–189
  using with `switch` statements, 102, 103

- Breakpoints, setting for debugging, 108
- Brute-force algorithm, 851
- BST (binary search trees). *see* Binary search trees (BST)
- `BST` class
  - `AVLTree` class as subclass of, 999
  - `BST.java` example, 968–973
  - overview of, 965–966
  - `TestBSTDelete.java` example, 977–980
  - `TestBST.java` example, 973–974
  - time complexity of, 978
  - `Tree.java` example, 966–967
- Bubble sorts, 283
  - algorithms, 891
  - `BubbleSort.java` example, 891–892
  - overview of, 890–891
  - time complexity of, 892
- Buckets
  - bucket sorts, 907–909
  - separate chaining and, 1023–1024, 1042
- `BufferedInputStream` and `BufferedOutputStream` classes, 701–704
- Bugs (logic errors), 21, 108
- Bus, function of, 2–3
- Button, 646–648
- Button, `ButtonDemo.java`, 647–648
- `ButtonBase`, 646–647
- `ButtonDemo.java`, 647–648
- `byte` type, numeric types
  - hash codes for primitive types, 1017
  - overview of, 45
- Bytecode
  - translating Java source file into, 15
  - verifier, 17–18
- Bytes
  - defined, 3
  - measuring storage capacity in, 4

C

- C++, high-level languages, 8
- C, high-level languages, 8
- Cable modems, 6
- `Calendar` class, 507–508
- Call stacks
  - displaying in debugging, 108
  - invoking methods and, 210
- Calling
  - methods, 208–210
  - objects, 333
- `canRead` method, `File` class, 478–479
- `canWrite` method, `File` class, 478–479
- `capacity()` method, `StringBuilder` class, 397
- Case sensitivity
  - identifiers and, 40
  - in `Welcome.java`, 12
- Casting. *see* Type casting
- Casting objects
  - `CastingDemo.java` example, 430–431
  - overview of, 429–430
- Catching exceptions. *see also* `try-catch` blocks
  - `catch` block omitted when `finally` clause is used, 471
  - `CircleWithException.java` example, 467
  - `InputMismatchExceptionDemo.java` example, 458–459
  - overview of, 463–465
  - `QuotientWithException.java` example, 456–458
- CDs (compact discs), as storage device, 5
- Cells
  - in Sudoku grid, 300
  - in tic-tac-toe case study, 671–676
- Celsius, converting Fahrenheit to/from, 237–238
- Chained exceptions, 473–474
- `char` data type. *see* Characters (`char` data type)
- Characters (`char` data type)
  - applying numeric operators to, 226
  - in binary I/O, 698–699
  - case study: ignoring nonalphanumeric characters when checking palindromes, 398–400
  - casting to/from numeric types, 128–129
  - comparing, 78
  - comparing and testing, 129–131
  - constructing strings from arrays of, 388–389
  - converting to strings, 392
  - decimal and hexadecimal equivalents of ASCII character set, 1165
  - escape characters, 128
  - finding, 137–138
  - generic method for sorting array of `Comparable` objects, 758
  - hash codes for primitive types, 1017
  - overview of, 126
  - `RandomCharacter.java`, 226
  - retrieving in strings, 132
  - `TestRandomCharacter.java`, 226–227
  - Unicode and ASCII and, 127
- `charAt` (index) method
  - retrieving characters in strings, 132
  - `StringBuilder` class, 397–398
- CheckBox, 648–651
- `CheckBoxDemo.java`, 649–651
- Checked exceptions, 461
- `checkIndex` method, 935
- Checkpoint Questions, recurrence relations and, 847
- Child, in BST, 961–962
- Choice lists. *see* ComboBox
- `Circle` and `Ellipse`
  - overview, 572–573
  - `ShowEllipse.java`, 573–574
- `Circle` class, 324, 325
- Circular
  - doubly linked lists, 948
  - singly linked lists, 948
- Clarity, class design guidelines, 532
- Class diagrams, UML, 325
- Class loaders, 17
- Class modifiers, Java modifiers, 1168–1169
- `ClassCastException`, 430
- Classes
  - abstract. *see* Abstract classes
  - abstraction and encapsulation in, 368–372
  - benefits of generics, 752–754
  - case study: designing class for matrix using generic types, 766–771
  - case study: designing class for stacks, 380–382
  - case study: designing `Course` class, 378–380
  - in `Circle.java` (for `CircleWithPrivateDataFields`), 347–348
  - in `Circle.java` (for `CircleWithStaticMembers`) example, 340–341
  - clients of, 327
  - commenting, 18
  - in `ComputeExpression.java`, 14–15
  - data field encapsulation for maintaining, 346–347
  - defining custom exception classes, 474–477
  - defining for objects, 324–326
  - defining generic, 754–756
  - design guidelines, 531–534
  - identifiers, 40
  - inner (nested) classes. *see* Inner (nested) classes
  - from Java Library, 336–339
  - names/naming conventions, 12, 44

Classes (*Continued*)
  Point2D, 338–339
  preventing extension of, 445
  raw types and backward compatibility, 760–761
  static variables, constants, and methods, 339–344
  in `TestCircleWithPrivateDataFields.java` example, 348–349
  in `TestCircleWithStaticMembers.java` example, 341–344
  in UML diagram, 325, 326
  variable scope and, 339–340
  visibility modifiers, 344–346
  in `Welcome.java`, 12
  in `WelcomeWithThreeMessages.java`, 14
Classes, binary I/O
  `BufferedInputStream` and `BufferedOutputStream` classes, 701–704
  `DataInputStream` and `DataOutputStream` classes, 698–701
  `DetectEndOfFile.java`, 701
  `FileInputStream` and `FileOutputStream` classes, 695–698
  `FilterInputStream` and `FilterOutputStream` classes, 698
  overview of, 694–695
  `TestDataStream.java`, 699–700
  `TestFileStream.java`, 696–698
Classifier, 1147
Class's contract, 368
Clock speed, CPUs, 3
`ClockPane` Class
  `ClockPane.java`, 582–584
  `DisplayClock.java`, 581–582
  `paintClock` method, 583–584
`clone` method, shallow and deep copies, 520–521
`Cloneable` interface
  `House.java` example, 519–523
  overview, 518–519
Closest pair problem, two-dimensional array applied to, 298–300
Closest-pair animation, 883
Cloud storage, 5
Cluster, 1020
COBOL, high-level languages, 8
Code
  arrays for simplifying, 254–255
  comments and, 103
  incremental development, 164
  programming. *see* Programs/programming
  reuse. *see* Reusable code
  sharing. *see* Sharing code
  in software development process, 61
Coding trees, 985–990. *see also* Huffman coding trees
Coherent purpose, class design guidelines, 531–532
`collect` method, stream reduction using, 1144–1147
Collections
  `Collection` interface, 776–780
  `forEach` method, 782
  iterators for traversing collections, 780–781
  singleton and unmodifiable, 835
  static methods for, 792–795
  `TestCollection.java` example, 778–780
`Collections` class
  singleton and unmodifiable collections, 835
  static methods, 792
Collections Framework hierarchy
  `ArrayList` and `LinkedList` class, 784–787
  case study: displaying bouncing balls, 795–798
  case study: stacks used to evaluate expressions, 803–807
  `Collection` interface, 776–778
  `Comparator` interface, 787–791
  `Dequeue` interface, 800–803
  designing complex data structures, 1054
  `forEach` method, 782
  iterators for traversing collections, 780–781
  key terms, 807
  `List` interface, 783–787
  `Map` interface, 1016
  methods of `List` interface, 783–787
  overview of, 776
  `PriorityQueue` class, 802–803
  `Queue` interface, 800
  queues and priority queues, 800–803
  quiz and exercises, 808–814
  static methods for lists and collections, 792–795
  summary, 808
  `TestCollection.java` example, 778–780
  `TestIterator.java` example, 780–781
  `Vector` and `Stack` classes, 798–800
Collisions, in hashing
  double hashing, 1022–1023
  handling using open addressing, 1019–1023
  handling using separate chaining, 1023–1025
  linear probing, 1019–1020
  overview of, 1017
  quadratic probing, 1020–1021
Column index, 291
ComboBox
  `ComboBoxDemo.java`, 660–662
  overview of, 659–660
Command-line arguments, 276–279
Comments
  code maintainability and, 103
  programming style and, 18
  in `Welcome.java`, 13
Common denominator, finding greatest common denominator. *see* Gcd (greatest common denominator)
Communication devices, computers and, 6
Compact discs (CDs), as storage device, 5
`Comparable` interface
  `ComparableRectangle.java` example, 515–516
  `Comparator` interface *vs.*, 788–789
  enumerated types, 1181
  as example of generic interface, 752–753
  generic method for sorting array of `Comparable` objects, 758
  overview of, 514–515
  `PriorityQueue` class and, 802
  `Rational` class implementing, 528
  `SortComparableObjects.java` example, 515
  `SortRectangles.java` example, 516–517
  `TreeMap` class and, 831
`Comparator` interface
  `Comparable` *vs.*, 789
  `GeometricObjectComparator.java`, 787–788
  methods of, 787–788
  `PriorityQueue` class and, 802–803
  `sorted` method, 1134
  `SortStringByLength.java`, 789
  `SortStringIgnoreCase.java`, 790–791
  `TestComparator.java`, 788
  `TestTreeSetWithComparator.java` example, 821–823
  `TreeMap` class and, 831
`compare` method, 787–788
`compareTo` method
  `Cloneable` interface and, 518
  `Comparable` interface defining, 514
  `ComparableRectangle.java` example, 515–516
  comparing strings, generic method for sorting array of Comparable objects, 759
  implementing in `Rational` class, 528

  wrapper classes and, 383–384
`compareToIgnoreCase` method, 136, 790
Comparison operators (`==`), 78, 434
Compatibility, raw types and backward compatibility, 760–761
Compile errors (Syntax errors)
  common errors, 13–14
  debugging, 108
  programming errors, 19–20
Compile time
  error detection at, 752
  restrictions on generic types, 765
  `Xlint:unchecked` error, 760
Compilers
  ambiguous invocation and, 223
  reporting syntax errors, 19–20
  translating Java source file into bytecode file, 15–16
  translating source program into machine code, 8, 9
Complete binary tree, 900, 906
Complete graphs, 1048
Completeness, class design guidelines, 533
Complex numbers, `Math` class, 538
Components
  `ListView`, 662–664
  `ListViewDemo.java`, 664–665
  quiz and exercises, 682–690
  `ScrollBar`, 665–667
  `ScrollBarDemo.java`, 666–667
  `SliderDemo.java`, 669–670
  sliders, 669
  summary, 681–682
  `TextArea`, 655–658
  `TextAreaDemo.java`, 658
  `TextFieldDemo.java`, 654–655
Composition, in designing stacks and queues, 950–951
Composition relationships
  aggregation and, 376–377
  between `ArrayList` and `MyStack`, 441–442
Compound expressions
  case study: stacks used to evaluate, 803–804
  `EvaluateExpression.java` example, 805–807
Compression
  data compression using Huffman coding, 985–990
  of hash codes, 1018–1019
  `HuffmanCode.java` example, 987–990
Compute expression, 14
Computers
  communication devices, 6–7
  CPUs, 3
  input/output devices, 5–6
  memory, 4
  OSs (operating systems), 9–10
  overview of, 2–3
  programming languages, 7–9
  storage devices, 4–5
`concat` method, 133
Concatenate strings, 36, 133
Conditional operators, 105–106
Connect four game, 315
Connected circles problem
  `ConnectedCircles.java`, 1073
  overview of, 1072–1073
Connected graphs, 1048
Consistency, class design guidelines, 532
Consoles
  defined, 12
  formatting output, 146–150
  input, 12
  output, 12
  reading input, 37–39
Constant time, comparing growth functions, 848
Constants
  class, 339–340
  declaring, 340
  identifiers, 40
  KeyCode constants, 613
  named constants, 43–44
  naming conventions, 44
  wrapper classes and, 383
Constructor chaining, 419–420
Constructor modifiers, 1168–1169
Constructors, 360
  in abstract classes, 502
  for `AVLTree` class, 1002–1003
  for `BMI` class, 373–374
  calling subclass constructors, 418
  creating `Date` class, 337
  creating objects with, 331
  creating `Random` objects, 337
  for `DataInputStream` and `DataOutputStream` classes, 699
  generic classes and, 756
  interfaces *vs.* abstract classes, 523
  invoking with `this` reference, 360
  for `Loan` class, 370–372
  object methods and, 324
  `private`, 346
  for `String` class, 388
  for `StringBuilder` class, 395–396
  in `TestCircle.java` example, 327, 328
  in `TV.java` example, 329
  UML diagram of, 326
  for `UnweightedGraph` class, 1058–1059
  for `WeightedGraph` class, 1092–1093
  wrapper classes and, 383
Containers
  creating data structures, 776
  maps as, 828
  removing elements from, 826
  storing objects in, 777
  types supported by Java Collections Framework, 776
`contains` method, 826, 827
`continue` statements, for controlling loops, 186–189
Contract, object class as, 324
Control, 545–548
Control units, CPUs, 3
Control variables, in `for` loops, 173–174
`ControlCircle.java`, 600–601
Conversion methods, for wrapper classes, 383
Convex hull
  finding for set of points, 867–869
  gift-wrapping algorithm applied to, 867–868
  Graham's algorithm applied to, 868–869
  string matching, 869–876
Copying
  arrays, 260–261
  files, 704–706
Core, of CPU, 3
`cos` method, trigonometry, 122–123
Cosine function, 589
`count` method, 1134–1135
Counter-controlled loops, 161
Coupon collector's problem, 284–285
`Course` class, 378–379
CPUs (central processing units), 3

Cubic time, comparing growth functions, 848
Curly braces { }
  in block syntax, 13
  dangers of omitting, 174–175
  forgetting to use, 85–86
Cursor, mouse, 6
Cycle, connected graphs, 1048

D

.dat files (binary), 694
Data, arrays for referencing, 250
Data compression
  Huffman coding for, 985–990
  `HuffmanCode.java` example, 987–990
Data fields
  accessing object data, 333
  encapsulating, 346–349, 532
  in interfaces, 512
  object state represented by, 324
  referencing, 334, 358–360
  in `TestCircle.java` example, 327
  in `TV.java` example, 329
  UML diagram of, 326
Data modifiers, 1168–1169
Data streams. *see* `DataInputStream`/`DataOutputStream` classes
Data structures. *see also* Collections Framework hierarchy
  array lists. *see* `ArrayList` class
  choosing, 776
  collections. *see* Collections
  first-in, first-out, 800
  linked lists. *see* `LinkedList` class
  lists. *see* Lists
  queues. *see* Queues
  stacks. *see* Stacks
Data structures, implementing
  array lists, 928–935
  `GenericQueue.java` example, 951
  implementing `MyLinkedList` class, 939–947
  linked lists, 935–949
  lists, 924–928
  `MyArrayList` compared with `MyLinkedList`, 947
  `MyArrayList.java` example, 929–933
  `MyLinkedList` class, 924–925, 937, 947
  `MyPriorityQueue.java` example, 953–954
  overview of, 924
  priority queues, 953–954
  quiz and exercises, 955–957
  stacks and queues, 949–953
  summary, 955
  `TestMyArrayList.java` example, 933–935
  `TestMyLinkedList.java` example, 938–939
  `TestPriorityQueue.java` example, 954
  `TestStackQueue.java` example, 952–953
  variations on linked lists, 948–949
Data types
  ADT (Abstract data type), 368
  `boolean`, 78–80, 337
  `char`. *see* Characters (`char` data type)
  `double`. *see* double (double precision), numeric types
  `float`. *see* Floating-point numbers (`float` data type)
  fundamental. *see* Primitive types (fundamental types)
  generic. *see* Generics
  `int`. *see* Integers (`int` data type)
  `long`. *see* Long, numeric types
  numeric, 45–48
  reference types. *see* Reference types
  specifying, 35
  strings, 131
  types of, 41
  using abstract class as, 504
`DataInputStream`/`DataOutputStream` classes
  `DetectEndOfFile.java`, 701
  external sorts and, 909
  overview of, 698
  `TestDataStream.java`, 699–700
`Date` class
  case study: `Calendar` and `GregorianCalendar` classes, 507–508
  `java.util`, 336
De Morgan's law, 97
Debugging
  benefits of stepwise refinement, 234
  code modularization and, 217
  selections, 108
Decimal numbers
  `BigDecimal` class, 387–388
  converting to hexadecimals, 184–186, 219–221, 745
  converting to/from binary, 745, 1172
  converting to/from hexadecimal, 1173
  equivalents of ASCII character set, 1165
  overview of, 1171
Declaring constants, 43, 340
Declaring exceptions
  `CircleWithException.java` example, 467
  `ReadData.java` example, 483–484
  `TestCircleWithCustomException.java` example, 475–476
  `throws` keyword for, 462, 467
Declaring methods
  generic methods, 757
  static methods, 339
Declaring variables
  array variables, 250
  overview of, 40–41
  specifying data types and, 35–36
  two-dimensional array variables, 290–291
Decrement (`--`) operator, 57–58
Deep copies, 521
Default field values, for data fields, 334
Degree of vertex, 1048
Delete key, on keyboards, 6
`delete` method, `AVLTree` class, 1007, 1011
Delimiters, token reading methods and, 484
Denominator. *see* Gcd (greatest common denominator)
Denominators, in rational numbers, 526
Depth-first searches (DFS)
  applications, 1071–1072
  case study: connected circles problem, 1072–1073
  finding DFS trees, 1048
  implementing, 1069–1071
  traversing graphs, 1067–1068
Depth-first traversal, tree traversal, 964
`Deque` interface, `LinkedList` class, 800–803
`dequeue` method, 953
`DescriptionPane` class, 657–658
Descriptive names
  benefits of, 40
  for variables, 35
Deserialization, of objects, 709
Design guidelines, classes, 531–534
Determining Big *O*

  for repetition statements, 842–845
  for selection statements, 842–845
  for sequence statements, 842–845
DFS (depth-first searches). *see* Depth-first searches (DFS)
Dial-up modems, 6
Dictionaries, 1016. *see also* Maps
Digital subscriber lines (DSLs), 6
Digital versatile disc (DVDs), 5
Digits, matching, 100
Dijkstra's single-source shortest-path algorithm, 1110
Direct recursion, 723
Directed graphs, 1047
Directories
  case study: determining directory size, 731–732
  `DirectorySize.java`, 731–732
  `File` class and, 478
  file paths, 477
`disjoint` method, 794
Disks, as storage device, 5
Display message
  in `Welcome.java`, 12
  in `WelcomeWithThreeMessages.java`, 14
`distinct` method, 1132, 1134–1135
Divide-and-conquer algorithm, 861–864
Divide-and-conquer strategy. *see* Stepwise refinement
Division (`/=`) assignment operator, 42
Division operator (`/`), 46, 53
Documentation, programming and, 18
Dot operator (`.`), 333
Dot pitch, measuring sharpness of displays, 6
`double` (double precision), numeric types
  converting characters and numeric values to strings, 392
  declaring variables and, 41
  generic method for sorting array of `Comparable` objects, 758
  hash codes for primitive types, 1017
  `java.util.Random`, 337
  overview of numeric types, 45
  precision of, 181
Double hashing, collision handling, 1022–1023
`Double.parseDouble` method, 138
`DoubleStream`, 1136–1139
Doubly linked lists, 948
  deciding when to use, 176–178
  `do-while` loops, 171–173
`do-while` loops, 171–173
Downcasting objects, 429
`drawArc` method, 575–577
Drives, 5
Drop-down lists. *see* ComboBox
DSLs (digital subscriber lines), 6
DVDs (Digital versatile disc), 5
Dynamic binding, inheritance and, 425–429
Dynamic programming
  computing Fibonacci numbers, 849–851
  definition, 850
  Dijkstra's algorithm, 1110
  `ImprovedFibonacci.java` example, 850–851

E

Eclipse
  built in debugging, 108
  creating/editing Java source code, 15
`Edge` arrays
  representing edges, 1051
  weighted edges using, 1093–1094
`Edge` class, 1051
Edges
  adjacency lists, 1052–1054
  adjacency matrices, 1052
  adjacent and incident, 1048
  defining as objects, 1051
  `Graph.java` example, 1056
  on graphs, 1047
  Prim's algorithm and, 1103
  representing edge arrays, 1051
  `TestGraph.java` example, 1056
  `TestMinimumSpanningTree.java`, 1107–1108
  `TestWeightedGraph.java`, 1100–1101
  weighted adjacency matrices, 1094
  weighted edges using edge array, 1093–1094
  weighted graphs, 1093
  `WeightedGraph` class, 1091
Edge-weighted graphs
  overview of, 1093
  `WeightedGraph` class, 1091
Eight Queens puzzle
  `EightQueens.java` example, 865–867
  recursion, 748
  single-dimensional arrays, 288
  solving, 864–867
Element type, specifying for arrays, 250
Emirp, 243
Encapsulation
  in `Circle.java` (for `CircleWithPrivateDataFields`) example, 347–348
  class design guidelines, 530
  of classes, 368–372
  of data fields, 346–349
  information hiding with, 227
  of `Rational` class, 530
Encoding schemes
  defined, 3–4
  mapping characters to binary equivalents, 127
End of file exception (`EOFException`), 701
End-of-line style, block styles, 19
`enqueue` method, 953
`entrySet` method, `Map` interface, 830
Enumerated types
  with data fields, constructors, and methods, 1184–1185
  `EnumeratedTypeDemo.java` example, 1182
  `if` statements with, 1183
  simple, 1181–1183
  `StandaloneEnumTypeDemo.java` example, 1182–1183
  `switch` statements with, 1183
  `TestTrafficLight.java` example, 1184–1185
  `TrafficLight.java` example, 1184
  `values` method, 1184
Equal (`=`) operator, for assignment, 78
== (equal to operator), 78
Equal to (`==`) operator, for comparison, 76
`equalArea` method, for comparing areas of geometric objects, 503
Equals method
  `Arrays` class, 275
  `Comparator` interface, 787
  `Object` class, 424
Erasure and restrictions, on generics, 764–766
`Error` class, 460, 462
Errors, programming. *see also* Programming errors
  (escape characters), 128

- Euclid's algorithm
  - finding greatest common divisors using, 851–855
  - `GCDEuclid.java` example, 852–855
- Euler, 1046–1047
- Event delegation, 597
- Event handlers/event handling, 594–595, 605–607, 613
- `Exception` class
  - exceptions in, 460
  - extending, 474
  - in `java.lang`, 474
  - subclasses of, 460–461
- Exception handling. *see also* Programming errors
  - catching exceptions, 463–465, 467–470
  - chained exceptions, 473–474
  - `ChainedExceptionDemo.java` example, 473–474
  - checked and unchecked, 461
  - `CircleWithException.java` example, 467
  - `ClassCastException`, 430
  - declaring exceptions (`throws`), 462, 467
  - defined, 454
  - defining custom exception classes, 474–477
  - `EOFException`, 701
  - in `Exception` class, 460
  - exception classes cannot be generic, 766
  - `FileNotFoundException`, 695
  - files input/output, 480–486
  - finally clause in, 470–471
  - getting information about exceptions, 465–467
  - in `House.java` example, 521
  - `InputMismatchExceptionDemo.java` example, 458–459
  - `InvalidRadiusException.java` example, 474–475
  - `IOException`, 461
  - key terms, 491
  - `NotSerializableException`, 709
  - overview of, 454–459
  - quiz and exercises, 492–497
  - `Quotient.java` example, 454
  - `QuotientWithException.java` example, 456–458
  - `QuotientWithIf.java` example, 455
  - `QuotientWithMethod.java` example, 455–456
  - `ReadData.java` example, 483–484
  - `ReadFileFromURL.java` example, 487–488
  - `ReplaceText.java` example, 485–486
  - rethrowing exceptions, 472–473
  - summary, 491–492
  - `TestCircleWithCustomException.java` example, 475–477
  - `TestCircleWithException.java` example, 468–470
  - `TestException.java` example, 466–467
  - `TestFileClass.java` example, 479
  - throwing exceptions, 462–463, 467–470
  - types of exceptions, 459–461
  - unsupported operations of `Collection` interface, 778
  - `WebCrawler.java` example, 489–491
  - when to use exceptions, 472
  - `WriteData.java` example, 480–481
  - `WriteDataWithAutoClose.java` example, 481–482
- Exception propagation, 463
- Exclusive or (`^`) logical operator, 95–99
- Execution stacks. *see* Call stacks
- `exists` method, for checking file instances, 478
- Explicit casting, 58, 59, 429
- Exponent method, `Math` class, 123
- Exponential algorithms, 847, 877
- Expressions
  - assignment statements and, 42–43
  - behind the scene evaluation, 107
  - Boolean. *see* Boolean expressions
  - case study: stacks used to evaluate, 803–804
  - `EvaluateExpression.java` example, 805–807
- `extends` keyword, interface inheritance and, 524
- External sorts
  - complexity, 916
  - `CreateLargeFile.java` example, 909–910
  - implementation phases, 911–913
  - overview of, 909

F

- Factorials
  - case study: computing factorials, 720–723
  - `ComputeFactorial.java`, 721–723
  - `ComputeFactorialTailRecusion.java`, 741
  - tail recursion and, 740–741
- `FadeTransition`, 622–623
- Fahrenheit, converting Celsius to/from, 237–238
- Fall-through behavior, `switch` statements, 103
- Feet, converting to/from meters, 238
- `fib` method, 724–726
- Fibonacci, Leonardo, 724
- Fibonacci numbers
  - algorithm for finding, 849–851
  - case study: computing, 723–726
  - `ComputeFibonacci.java`, 724–726
  - computing recursively, 743
  - `ImprovedFibonacci.java` example, 850–851
  - recurrence relations and, 847
- `File` class, 477–479, 692
- File I/O. *see* Input; Output
- File pointers, random-access files and, 712
- `FileInputStream`/`FileOutputStream` classes
  - overview of, 695–696
  - `TestFileStream.java`, 696–698
- Files
  - case study: copying files, 704–706
  - case study: replacing text in, 485–486
  - `File` class, 477–479, 692
  - input/output, 480–486
  - key terms, 491
  - quiz and exercises, 492–497
  - reading data from, 482–483
  - reading data from Web, 487–488
  - summary, 491–492
  - `TestFileClass.java` example, 479
  - writing data to, 480–481
- `fill` method, 794
- `filter` method, 1132, 1134
- `FilterInputStream`/`FilterOutputStream` classes, 698
- `final` keyword, for declaring constants, 43
- `final` modifier, for preventing classes from being extended, 445
- `finally` clause, in exception handling, 470–471
- `findAny` method, 1133, 1135–1136
- `findFirst` method, 1133, 1135–1136
- First-in, first-out data structures, 800
- `float` data type. *see* Floating-point numbers (`float` data type)
- Floating-point numbers (`float` data type)
  - approximation, 67
  - converting to integers, 58
  - hash codes for primitive types, 1017
  - `java.util.Random`, 337
  - minimizing numeric errors related to loops, 180–181
  - numeric types for, 45
  - overview of numeric types, 45

  - special values, 1170
  - specifying data types, 35
  - specifying precision, 148
- Flowcharts
  - `do-while` loop, 171
  - `if` statements and, 80–81
  - `if-else` statements, 82–83
  - `for` loops, 173, 174
  - `switch` statements, 102–103
  - `while` loops, 160, 161
- `FlowPane`
  - `HBox` and `VBox`, 564–565
  - overview, 559
  - `ShowFlowPane.java`, 559–561
- Folding, hash codes and, 1017
- Font, `FontDemo.java`, 554–555
- `for` loops
  - deciding when to use, 176–178
  - nesting, 178–180, 293
  - overview of, 173–176
  - processing arrays with, 253
  - variable scope and, 224–225
- foreach (enhanced) loops
  - implicit use of iterator by, 786
  - overview of, 255–257
  - for traversing collections, 781
- `forEach` method, 1133
- Formal generic type, 752
- Formal parameters. *see* Parameters
- Format specifiers, 147–149
- FORTRAN, high-level languages, 8
- Forward pointer, in doubly linked lists, 948
- Fractals
  - case study, 736–739
  - H-tree fractals, 749
  - Koch snowflake fractal, 747
  - `SierpinskiTriangle.java`, 736–739
- Frames (windows)
  - `ScrollBarDemo.java`, 666–667
  - `SliderDemo.java`, 669–670
- Free cells, in Sudoku grid, 300
- `frequency` method, 794
- Function keys, on keyboards, 6
- Functions, 207. *see also* Methods
- Fundamental types (Primitive types). *see* Primitive types (fundamental types)

G

- Galton box, 288
- Garbage collection, JVM and, 335
- GBs (gigabytes), of storage, 4
- Gcd (greatest common denominator)
  - algorithm for finding, 851–855
  - case study: finding greatest common denominator, 182–183
  - computing recursively, 742
  - `gcd` method, 217–218
  - `GCDEuclid.java` example, 853–855
  - `GCD.java` example, 852–853
  - `Rational` class and, 526–527
- Generic instantiation, 752
- Generics
  - case study: designing class for matrix using generic types, 766–771
  - case study: generic method for sorting array, 758–759
  - defining generic classes and interfaces, 754–756
  - erasing generic types, 764–766
  - `GenericStack` class, 755–756
  - key terms, 771
  - methods, 756–758
  - motivation for using, 752–754
  - overview of, 752
  - questions and exercises, 772–773
  - raw types and backward compatibility and, 760–761
  - restrictions on generic types, 764–766
  - summary, 771–772
  - wildcards for specifying range of generic types, 761–764
- `GeometricObject` class
  - `Circle.java` and `Rectangle.java`, 502
  - overview of, 501
  - `TestGeometricObject.java` example, 502–503
- `getAbsolutePath(), File` class, 479
- `getArea` method, `Circle` example, 327, 328
- `getArray` method, 295–296
- `getBMI` method, `BMI` class, 373, 374
- `getCharacterFrequency` method, 988
- `getChars` method, converting strings into arrays, 392
- `getDateCreated` method, `Date` class, 356
- `getIndex` method, `ArrayList` class, 437
- `getMinimumSpanningTree` method, `WeightedGraph` class, 1103–1104
- `getPerimeter` method, `Circle` example, 327
- `getRadius` method, `CircleWithPrivateDataFields.java` example, 348
- `getRandomLowerCaseLetter` method, 265, 267
- `getSize` method, finding directory size, 442, 1154–1155
- `getSource` method, events, 596
- `getStackTrace` method, for getting information about exceptions, 466
- `getStatus` method, BMI class, 373, 374
- Getter (accessor) methods
  - `ArrayList` class and, 438
  - encapsulation of data fields and, 347–348
  - implementing linked lists, 935
- Gift-wrapping algorithm, 867–868
- Gigabytes (GBs), of storage, 4
- Gigahertz (GHz), clock speed, 3
- GMT (Greenwich Mean Time), 54
- Gosling, James, 10
- Graham's algorithm, 868–869
- `Graph` interface, 1054
- Graph theory, 1046
- Graphical user interface (GUI), 644
- Graphs
  - breadth-first searches (BFS), 1074–1075
  - case study: connected circles problem, 1072–1074
  - case study: nine tails problem, 1077–1083
  - `ConnectedCircles.java`, 1073
  - `DisplayUSMap.java` example, 1066–1067
  - `Graph.java` example, 1056–1057
  - `GraphView.java` example, 1064–1065
  - key terms, 1083
  - modeling, 1054–1064
  - overview of, 1046–1047
  - questions and exercises, 1083–1089
  - representing edges, 1051–1054
  - representing vertices, 1048–1050
  - summary, 1083
  - terminology regarding, 1047–1048
  - `TestGraph.java` example, 1056–1057
  - traversing, 1067
  - `UnweightedGraph.java` example, 1058–1064
  - visualization of, 1064–1067
- Greater than (`>`) comparison operator, 78
- Greater than or equal to (`>=`) comparison operator, 78

Greatest common denominator. *see* Gcd (greatest common denominator)
Greedy algorithms
  Dijkstra's algorithm, 1109
  overview of, 986
Greenwich Mean Time (GMT), 54
`GregorianCalendar` class
  `Cloneable` interface and, 518–519
  in `java.util` package, 363
  overview of, 507–508
  `TestCalendar.java` example, 508–510
`GridPane`
  overview, 561–562
  `ShowGridPane.java`, 562–563
Grids, representing using two-dimensional array, 300
Group classifier, 1147
Group processor, 1147
`groupingby` collector, grouping elements using, 1147–1150
Growth rates
  algorithm for comparing, 847–848
  comparing algorithms based on, 840

H

Hamiltonian path/cycle, 1087
`HandleEvent.java`, 595–596
Hand-traces, for debugging, 108
Hangman game, 287, 495, 589, 809, 810
Hard disks, as storage device, 5
Hardware, 2
Has-a relationships
  in aggregation models, 376–377
  composition and, 442
Hash codes, 1017
  compressing, 1018–1019
  vs. hash functions, 1017–1019
  for primitive types, 1017
  for strings, 1017–1018
Hash functions, 1016
  vs. hash codes, 1017–1019
  as index to hash table, 1016
Hash map, 831, 1034
Hash set, 816
Hash tables, 1016, 1035. *see also* Maps
  measuring fullness using load factor, 1024–1025
  parameters, 1031
`hashCode` method, 816, 1017
Hashing
  collisions handling using open addressing, 1019–1023
  collisions handling using separate chaining, 1023
  compressing hash codes, 1018–1019
  double hashing open addressing, 1022–1023
  function, 1016–1019
  hash codes for primitive types, 1017
  hash codes for strings, 1017–1018
  hash functions *vs.* hash codes, 1017–1019
  key terms, 1041–1042
  linear probing open addressing, 1019–1020
  load factor and rehashing, 1025
  map implementation with, 1025–1034
  `MyHashMap.java` example of map implementation, 1027–1033
  `MyHashSet.java` example of set implementation, 1034–1041
  `MyMap.java` example of map implementation, 1026–1027
  overview of, 1016
  quadratic probing open addressing, 1020–1021
  quiz and exercises, 1042–1044
  set implementation with, 1034–1041
  summary, 1042
  `TestMyHashMap.java` example of map implementation, 1033–1034
  `TestMyHashSet.java` example of set implementation, 1041
  what it is, 1016–1017
`HashMap` class
  concrete implementation of `Map` class, 828–830
  implementation of `Map` class, 1016
  load factor thresholds, 1025
  overview of, 831
  `TestMap.java` example, 831–833
  types of maps, 829
`HashSet` class
  case study: counting keywords, 827–828
  implementation of `Set` class, 1034
  overview of, 816–820
  `TestHashSet.java` example, 817–818
  `TestMethodsInCollection.java` example, 818–819
  types of sets, 816
`Hashtable`, 831
`HBox` and `VBox`
  definition, 566
  overview, 564
  `ShowHBoxVBox.java`, 565–566
`Heap` class
  `Heap.java` example, 904–905
  operations for manipulating heaps in, 904
  sorting arrays with, 906
Heap sorts, 900–907
  adding nodes to heaps, 901–902
  arrays using heaps, 906
  `Heap` class, 903–904
  `Heap.java` example, 904–905
  `HeapSort.java` example, 906
  overview of, 900–901
  removing root from heap, 902–903
  storing heaps, 901
  time complexity of, 906–907
Heaps
  adding nodes to, 901–902
  arrays using, 906
  binary heaps (binary trees), 900
  dynamic memory allocation and, 263
  height of, 906
  implementing priority queues with, 953–954
  removing root from, 902–903
  storing, 901
Height, 960
Height of a heap, 906
Helper method, recursive
  overview of, 728
  `RecursivePalindrome.java`, 728–729
Hertz (Hz), clock speed in, 3
Hexadecimal numbers
  converting to/from binary, 1173–1174
  converting to/from decimal, 143–144, 184–186, 219–221, 745, 1173
  equivalents of ASCII character set, 1165
  overview of, 1171
Hidden data fields, 358, 360
High-level languages, 8–9
Hilbert curve, 749
Horizontal scroll bars, 666
Horizontal sliders, 668, 669
H-trees
  fractals, 749
  recursive approach to, 720
Huffman coding, 985–990

- Huffman coding trees
  - data compression using, 985–990
  - `HuffmanCode.java` example, 987–990
- Hz (Hertz), clock speed in, 3

I

- Identifiers, 40
- IDEs (integrated development environments) for creating/editing Java source code, 11, 12, 15–16
- IEEE (Institute of Electrical and Electronics Engineers), floating point standard (IEEE 754), 45
- `if` statements
  - common errors, 85–89
  - in computing body mass index, 91–92
  - in computing taxes, 92–95
  - conditional operator used with, 105–106
  - with enumerated types, 1183
  - nesting, 83–85
  - overview of, 80–82
  - `SimpleIfDemo.java` example, 81–82
- `if-else` statements
  - conditional expressions and, 105
  - dangling else ambiguity, 87
  - multi-way, 83–85
  - overview of, 82–85
  - recursion and, 726
- `IllegalArgumentException` class, 463
- `Image`, 556–558
- `Image` class, 556
- Image icons, `ComboBoxDemo.java`, 660
- Images, `ShowImage.java`, 557–558
- `ImageView`, 556–558
- Immutable
  - `BigInteger` and `BigDecimal` classes, 387–388
  - class, 355
  - objects, 355–356
  - `Rational` class, 530
  - `String` object, 388–389
  - wrapper classes, 383
- Implementation (coding), in software development process, 63–64
- Implementation methods, 231–234
- Implicit casting, 129, 429
- Importing, types of `import` statements, 38
- Increment method, in `Increment.java` example, 214
- Increment (`++`) operator, 57–58
- Incremental development
  - benefits of stepwise refinement, 234
  - coding incrementally, 164
- Indentation, programming style, 19
- Indexed variables elements, 252
- Indexes
  - accessing elements in arrays, 250, 252
  - finding characters/substrings in a string, 137–138
  - `List` interface and, 783, 784
  - `MyList.java` example, 926–928
  - string index range, 133
- `indexOf` method, 137–138
  - implementing `MyLinkedList`, 946
  - `List` interface, 783
  - `MyArrayList.java` example, 926, 931, 933
- Indirect recursion, 723
- Infinite loops, 162
- Infinite recursion, 723
- Information
  - getting information about exceptions, 465–467
  - hiding (encapsulation), 227
- Inheritance
  - `ArrayList` object, 434–435
  - calling subclass constructors, 418
  - calling superclass methods, 420–421
  - case study: custom stack class, 441–442
  - casting objects and, 429–433
  - `CastingDemo.java` example, 430–431
  - `Circle.java` example, 414–416
  - constructor chaining and, 419–420
  - in designing stacks and queues, 950
  - `DistinctNumbers.java` example, 438–440
  - dynamic binding and, 425–429
  - `equals` method of `Object` class, 433–434
  - generic classes, 756
  - `GeometricObject.java` example, 413–414
  - interface inheritance, 510–511, 524
  - is-a relationships and, 442
  - key terms, 445
  - `Object` class and, 424
  - overriding methods and, 421–422
  - overview of, 412
  - preventing classes from being extended or overridden, 445
  - `protected` data and methods, 442–444
  - quiz and exercises, 447–451
  - `Rectangle.java` example, 416–417
  - summary, 446
  - superclasses and subclasses and, 412–418
  - `TestArrayList.java` example, 435–438
  - `TestCircleRectangle.java` example, 417–418
  - using`super` keyword, 418
- Initializing variables
  - `AnalyzeNumbers.java`, 257
  - arrays, 252
  - declaring variables and, 41
  - multidimensional arrays, 291
  - two-dimensional arrays, 293
- Inner (nested) classes
  - anonymous, 602–603
  - `AnonymousHandlerDemo.java`, 603–605
  - for defining listener classes, 601–602
  - `KeyEventDemo.java`, 614–615
  - `ShortestPathTree` class as inner class of `WeightedGraph` class, 1127–1128
  - `TicTacToe.java`, 672–676
- Inorder traversal
  - time complexity of, 978
  - tree traversal, 963
- Input
  - reading from console, 37–39
  - redirecting using `while` loops, 169–170
  - runtime errors, 21
  - streams. *see* `InputStream` classes
- Input, process, output (IPO), 39
- `InputMismatchException` class, 458, 484
- Input/output devices, computers and, 5–6
- `InputStream` classes
  - `BufferedInputStream`, 701–704
  - case study: copying files, 705–706
  - `DataInputStream`, 698–701
  - deserialization and, 709
  - `DetectEndOfFile.java`, 701
  - `FileInputStream`, 695–696
  - `FilterInputStream`, 698
  - `ObjectInputStream`, 706–707
  - overview of, 694–695

`InputStream` classes (*Continued*)
  `TestDataStream.java`, 699–700
  `TestFileStream.java`, 696–698
  `TestObjectInputStream.java`, 708
Insert key, on keyboards, 6
`insert` method
  `AVLTree` class, 1011
  overriding, 1000–1001
Insertion order, `LinkedHashMap` class, 831
Insertion sort algorithm
  analyzing, 846
  recurrence relations and, 847
Insertion sorts
  algorithms, 882–884
  `InsertionSort.java` example, 888–889
  time complexity of, 890
Instance methods
  accessing object data and methods, 333
  in `Circle.java` (for `CircleWithStaticMembers`), 340–341
  class design guidelines, 326–327
  invoking, 370, 373
  when to use instance methods *vs.* static, 341–344
Instance variables
  accessing object data and methods, 333
  class design guidelines, 343
  static variables compared with, 339–341
  in `TestCircleWithStaticMembers.java` example, 341
  when to use instance variables *vs.* static, 341–344
Instances. *see also* Objects
  checking file instantiation, 478
  checking object instantiation, 324, 430
  generic instantiation, 752
Institute of Electrical and Electronics Engineers (IEEE), floating point standard (IEEE 754), 45
`int` data type. *see* Integers (`int` data type)
`Integer.parseInt` method, 138
Integers (`int` data type)
  `ArrayList` for, 439
  `BigInteger` class, 387–388
  bit operators and, 1175
  case study: designing class for matrix using generic types, 766, 767
  casting to/from `char` types, 128–129
  converting characters and numeric values to strings, 392–393
  declaring variables and, 41
  division of, 454–458
  finding larger between two, 207
  floating-point numbers converted to, 56–57
  generic method for sorting array of Comparable objects, 758
  greatest common denominator of, 851
  hash codes for primitive types, 1017
  `IntegerMatrix.java` example, 769
  `java.util.Random`, 337
  numeric types for, 45
  sorting, 907
  sorting`int` values, 913
  specifying data types, 35
  `TestIntegerMatrix.java` example, 770
Integrated development environments (IDEs), 11, 12
  for creating/editing Java source code, 15–16
  overview of, 11
Intelligent guesses, 164
Interfaces
  abstract classes compared with, 521–523
  benefits of, 516
  benefits of generics, 752
  case study: `Rational` class, 526–527
  `Cloneable` interface, 518–519
  `Comparable` interface, 513–514
  `ComparableRectangle.java` example, 515–516
  for defining common class behaviors, 566
  defining generic, 754–756
  `House.java` example, 519–523
  key terms, 534
  overview of, 501
  questions and exercises, 535–540
  raw types and backward compartibility, 760
  `SortComparableObjects.java` example, 515
  `SortRectangles.java` example, 516–517
  summary, 534–535
  `TestEdible.java` example, 510–513
Intermediate method, `Stream` interface, 1130
Interned strings, 389
Interpreters, translating source program into machine code, 8, 9
`IntStream`, 1136–1139
Invoking methods, 208, 209, 333, 757
  binary I/O classes, 694–695
  `BufferedInputStream` and `BufferedOutputStream` classes, 701–704
  case study: copying files, 704–705
  case study: replacing text, 485–486
  `Copy.java`, 705–706
  `DataInputStream` and `DataOutputStream` classes, 698–701
  `DetectEndOfFile.java`, 701
  `FileInputStream` and `FileOutputStream` classes, 695–698
  `FilterInputStream` and `FilterOutputStream` classes, 698
  handling text I/O in Java, 692–693
  key terms, 714
  object I/O, 706–707
  overview of, 481, 692
  questions and exercises, 715–718
  random-access files, 711–714
  reading data from file using `Scanner` class, 482–484
  reading data from the Web, 487–488
  `serializable` interface, 708–709
  serializing arrays, 709–711
  summary, 715
  `TestDataStream.java`, 699–700
  `TestFileStream.java`, 696–698
  `TestObjectInputStream.java`, 708
  `TestObjectOutputStream.java`, 707–708
  `TestRandomAccessFile.java`, 713–714
  text I/O *vs.* binary I/O, 693–694
  types of I/O devices, 5–6
  writing data to file using `PrintWriter` class, 480–481
`IOException`, 695, 696
IPO (input, process, output), 39
Is-a relationships
  design guide for when to use interfaces *vs.* classes, 524
  inheritance and, 442
`isAbsolute` method, for checking file instances, 478–479
`isDigit` method, `Character` class, 144
`isDirectory` method, for checking file instances, 478–479
`isFile` method, for checking file instances, 478–479
`isHidden` method, for checking file instances, 478–479
Is-kind-of relationships, 524
`isPalindrome` method
  `RecursivePalindrome.java`, 728–729
  as tail-recursive method, 740
`isPrime` method, prime numbers, 219
`isValid` method, applying to Sudoku grid, 302
`Iterable` interface, 780

Iteration/iterators
  binary search trees and, 983–985
  implementing`MyLinkedList`, 944, 947
  `Iterable` interface, 984
  `Iterator` object, 780
  lists and, 780–781
  loops and, 160
  `MyArrayList.java` example, 932, 933
  recursion compared with, 740
  `TestIterator.java` example, 780–781
  `TestMyArrayList.java` example, 933–934
  traversing collections, 780–781

J

Java Collections Framework. *see* Collections Framework hierarchy
`java` command, for executing Java program, 11
Java Development Toolkit (JDK)
  jdb debugger in, 108
  overview of, 11, 12
Java EE (Java Enterprise Edition), 11
Java language specification, 11–12
Java Library, 336–339
Java ME (Java Micro Edition), 12
Java programming
  creating, compiling, and executing programs, 15–18
  displaying text in message dialog box, 23
  high-level languages, 8–9
  introduction to, 12
  simple examples, 12–15
  using Eclipse, 25–28
  usingNetBeans, 23–25
Java SE (Java Standard Edition), 11–12
Java Virtual Machine. *see* JVM (Java Virtual Machine)
`javac` command, for compiling Java program, 17
`Javadoc` comments (`/**.*/`), 18
JavaFX
  `Arc`, 575–577
  binding properties, 548–550
  `BorderPane`, 563–564
  case study: `ClockPane Class`, 580–584
  `Circle` and `Ellipse`, 572–574
  `Color` class, 553–554
  `FlowPane`, 559–561
  `Font` class, 554–555
  `GridPane`, 561–563
  `HBox` and `VBox`, 564–565
  `Image` and `ImageView` Classes, 556–558
  key terms, 585
  Layout panes, 558–559
  `Line`, 569–570
  nodes, 551–552
  panes, 545–548
  `Polygon` and `Polyline`, 577–580
  quiz and exercises, 586–591
  `Rectangle`, 570–572
  shapes, 567
  structure, 542–545
  summary, 585–586
  vs. Swing and AWT, 542
  `Text`, 567–569
JavaFX CSS, 551
JavaFX UI controls
  `ScrollBar`, 665–667
  `BounceBallSlider.java`, 670–671
  button, 646–648
  `ButtonDemo.java`, 647–648
  case study: developing tic-tac-toe game, 671–676
  case study: national flags and anthems, 679–681
  `CheckBox`, 648–651
  `CheckBoxDemo.java`, 649–651
  `ComboBox`, 659–662
  `ComboBoxDemo.java`, 660–662
  `DescriptionPane.java`, 657–658
  `Labeled` and `Label`, 644–646
  `LabelWithGraphic.java`, 644–646
  `ListView`, 662–664
  `ListViewDemo.java`, 664–665
  `MediaDemo.java`, 677–679
  programming exercises, 682–689
  quiz, 682
  `RadioButton`, 651–653
  `RadioButtonDemo.java`, 652–653
  `ScrollBar`, 665–667
  `ScrollBarDemo.java`, 666–667
  `Slider`, 668–671
  `SliderDemo.java`, 669–670
  `TextArea`, 655–658
  `TextAreaDemo.java`, 658
  `Textfield`, 654–655
  `TextFieldDemo.java`, 654–655
  `TicTacToe.java`, 672–676
  video and audio, 676–679
`java.io`
  `File` class, 477–479
  `PrintWriter` class, 480
  `RandomAccessFile` class, 712
`java.lang`
  `Comparable` interface, 514
  `Exception` class, 474
  `Number` class, 505
  packages, 64
  `Throwable` class, 459–461, 465
`java.net`
  `MalformedURLException` class, 487
  `URL` class, 486
`java.util`
  `Arrays` class, 272–274
  `Calendar` class, 507–508
  creating stacks, 806
  `Date` class, 336
  `EventObject` class, 596–597
  `GregorianCalendar` class, 363, 507–508
  Java Collections Framework and, 776
  `Random` class, 337
  `Scanner` class, 38, 482–484
jdb debugger, 108
JDK (Java Development Toolkit)
  jdb debugger in, 108
  overview of, 12
JShell, 50–52
JVM (Java Virtual Machine)
  defined, 16
  detecting runtime errors, 454
  garbage collection, 258
  heap as storage area in, 263
  interned string and, 389

K

KBs (kilobytes), 4
Key constants, 613

Keyboards, 5–6
`KeyEvents`
  `ControlCircleWithMouseAndKey.java`, 615–616
  `KeyEventDemo.java`, 614–615
  overview of, 613–614
Keys
  hashing functions, 1016
  maps and, 1042
`keySet` method, `Map` interface, 830
Key/value pairs, in maps, 828–829
Keywords (reserved words)
  `break` and `continue`, 186–189
  case study: counting, 827–828
  `extends`, 524
  `final`, 43
  list of Java keywords, 1163
  `super`, 418
  `throws`, 462, 463
  `transient`, 709
  in `Welcome.java`, 13
Kilobytes (KBs), 4
Knight's Tour, 747–748
Knuth-Morris-Pratt algorithm, 873–876
  `StringMatchKMP.java`, 875–876
Koch snowflake fractal, 747
Kruskal's algorithm, 1122

L

Label, 644–646
Labeled, 644–646
Labeling vertices, 1050
Labels, `LabelWithGraphic.java`, 644–646
Lambda expression
  `LambdaHandlerDemo.java`, 607–609
  overview of, 605–607
Landis, E. M., 996
LANs (local area networks), 6
`lastIndexOf` method
  implementing `MyLinkedList`, 946
  `List` interface, 783
  `MyArrayList.java` example, 931, 933
  `MyList.java` example, 926
  strings, 137–138
`lastModified` method, File class, 479
Latin square, 320–321
Layout panes
  `BorderPane`, 563–564
  `FlowPane`, 559
  `GridPane`, 561–563
  `HBox` and `VBox`, 564–565
Lazy operator, 98
Leaf, 960
  deleting, 975
Left subtree, of binary trees, 960
Left-heavy, balancing AVL nodes, 996, 1003
Length, 960
`length` method, for checking file instances, 478–479
Length, strings, 131–132, 397
Letters, counting, 265–266
Level, 960
Libraries, APIs as, 11
`Line`
  overview, 569
  `ShowLine.java`, 569–570
Line comments, in `Welcome.java`, 13
Line numbers, in `Welcome.java`, 12
Linear probing, collision handling, 1019–1020
Linear search algorithm, 882
  comparing growth functions, 848
  recurrence relations and, 847
Linear searches, arrays, 269–270
Linked data structures
  binary search trees, 960–961
  hash maps. *see* `LinkedHashMap` class
  hash sets. *see* `LinkedHashSet` class
  lists. *see* `LinkedList` class
Linked hash map, 831, 832
Linked hash set, 820, 825
`LinkedHashMap` class
  concrete implementation of `Map` class, 828–830
  implementation of `Map` class, 1016
  overview of, 831
  `TestMap.java` example, 831–833
  types of maps, 828–829
`LinkedHashSet` class
  implementation of `Set` class, 1034
  ordering elements in hash sets, 818
  overview of, 820
  `SetListPerformanceTest.java` example, 825–826
  types of sets, 816
`LinkedList` class
  animation of linked lists, 924, 925
  compared with `ArrayList` class, 784–786
  defined under `List` interface, 783
  `Deque` interface, 800–802
  implementing buckets, 1023
  implementing linked lists, 935–949
  implementing `MyLinkedList` class, 939–947
  implementing queues using linked lists. *see* Queues
  `MyArrayList` compared with `MyLinkedList`, 947
  `MyLinkedList`, 924–925, 937, 947
  representing edges in graphs using linked lists, 1054
  `SetListPerformanceTest.java` example, 825–826
  `TestArrayAndLinkedList.java`, 785–786
  `TestMyLinkedList.java` example, 938–939
  variations on linked lists, 948–949
Linux OS, 9
`List` interface
  common features of lists defined in, 924
  methods of, 783–784
  overview of, 783
  `Vector` class implementing, 799, 800
`ListIterator` interface, 783
Lists
  adjacency lists for representing edges, 1052–1054
  array lists. *see* `ArrayList` class
  as collection type, 776
  comparing performance with sets, 824–826
  implementing, 924–928
  linked lists. *see* `LinkedList` class
  `List` interface, 783–784
  `ListViewDemo.java`, 664–665
  methods of `List` interface, 783–784
  `MyList.java` example, 926–928
  singleton and unmodifiable, 835
  static methods for, 792–795
`ListView`, 662–664
Literal values, not using as identifiers, 1163
Literals
  Boolean literals, 79
  character literals, 126
  constructing strings from string literal, 388
  defined, 48

- floating-point literals, 49
- integer literals, 49

LL imbalance, AVL nodes, 996
LL rotation
- `AVLTree` class, 1002
- balancing nodes on a path, 1000
- implementing, 1001
- options for balancing AVL nodes, 996

Load factor
- hash sets and, 816
- rehashing and, 1025

`LoanCalculator.java`, 610–611
Loans
- Loan calculator case study, in event-driven programming, 609–611
- `Loan.java` object, 370–372

Local area networks (LANs), 6–7
Local variables, 224
Locker puzzle, 284
Logarithmic algorithm, 846
Logic errors (bugs), 21, 108
Logical operators (Boolean operators)
- overview of, 95
- `TestBooleanOperators.java` example, 96–99
- truth tables, 95–96

Long, numeric types
- converting characters and numeric values to strings, 392
- hash codes for primitive types, 1017
- integer literals and, 49
- `java.util.Random`, 337
- overview of numeric types, 45

`LongStream`, 1136–1139
Loop body, 160
Loop-continuation-condition
- `do-while` loop, 171
- loop design and, 166
- in multiple subtraction quiz, 166
- overview of, 160–161

Loops
- `break` and `continue` keywords as controls in, 186–189
- case study: displaying prime numbers, 191–193
- case study: finding greatest common denominator, 182–183
- case study: guessing numbers, 163–166
- case study: multiple subtraction quiz, 166–168
- case study: predicting future tuition, 183–184
- creating arrays, 261
- deciding which to use, 176–178
- design strategies, 166–168
- `do-while` loop, 171–173
- examples of determining Big *O*, 842–845
- graph edges, 1048
- input and output redirections, 170
- iteration compared with recursion, 740
- key terms, 193
- `for` loops, 173–176
- minimizing numeric errors related to, 180–181
- nesting, 178–180
- overview of, 160
- quiz and exercises, 194
- sentinel-controlled, 168–170
- summary, 193–194
- `while` loop, 160–163

Lottery game, 809
Lower-bound wildcards, 762
Low-level languages, 8
LR imbalance, AVL nodes, 997, 998
LR rotation
- `AVLTree` class, 1003
- balancing nodes on a path, 1000
- options for balancing AVL nodes, 997, 998

M

Mac OS, 9
Machine language
- bytecode compared with, 16
- overview of, 7
- translating source program into, 8, 9

Machine stacks. *see* Call stacks
Main class
- defined, 325
- in `TestCircle.java` example, 326

`main` method
- in `Circle.java (AlternativeCircle.java)` example, 328, 329
- in `ComputeExpression.java`, 14–15
- invoking, 210
- main class *vs.*, 325
- receiving string arguments from command line, 276–277
- in `TestCircle.java` example, 325
- in `TestTV.java` example, 330–331
- in `Welcome.java`, 12
- in `WelcomeWithThreeMessages.java`, 14

Maintenance, in software development process, 62
`MalformedURLException` class, 487
`Map` interface
- methods, 829–830, 832
- overview of, 829

`map` method, 1134–1135
Maps
- case study: counting occurrence of words using tree map, 833–834
- containers supported by Java Collections Framework, 776
- hash maps. *see* `HashMap` class
- key terms, 836
- linked hash maps. *see* `LinkedHashMap` class
- overview of, 816, 828–833
- quiz and exercises, 836–838
- singleton and unmodifiable, 835
- summary, 836
- `TestMap.java` example, 831–833
- tree maps. *see* `TreeMap` class

Maps, implementing with hashing
- `MyHashMap.java` example, 1027–1033
- `MyMap.java` example, 1026–1027
- overview of, 1025
- `TestMyHashMap.java` example, 1033–1034

`mapToInt` method, 1137, 1138
Marker interfaces, 518
Match braces, in `Welcome.java`, 12
matches method, strings, 391
`Math` class
- `BigInteger` and `BigDecimal` classes, 387
- case study: computing angles of a triangle, 125–126
- complex numbers, 538–539
- exponent methods, 123
- invoking object methods, 333
- methods generally, 122
- random method, 89–90, 100–101, 124
- rounding methods, 124
- service methods, 122
- trigonometric methods, 122

Matrices
- adjacency matrices for representing edges, 1052–1054
- case study: designing class for matrix using generic types, 766–767

Matrices (*Continued*)
  `GenericMatrix.java` example, 767–769
  `IntegerMatrix.java` example, 769
  `RationalMatrix.java` example, 769–770
  `TestIntegerMatrix.java` example, 770
  `TestRationalMatrix.java` example, 770–771
  two-dimensional arrays for storing, 290–291
`max and min` method, 1132, 1134
`max` method
  defining and invoking, 208–210
  finding minimum element in lists, 794
  `GeometricObjectComparator.java` example, 788
  `MaxUsingGenericType.java` example, 760–761
  overloading, 221
  overview of, 124
`maxRow` variable, for finding largest sum, 294
Mbps (million bits per second), 7
MBs (megabytes), of storage, 4
`Media`, 676–679
`MediaPlayer`, 676–679
`MediaView`, 676–679
Megabytes (MBs), of storage, 4
Megahertz (MHz), clock speed, 3
Memory, computers, 4
Merge sorts
  algorithms, 892
  heap sort compared with, 906
  merge sort algorithms, 892
  `MergeSort.java` example, 893–894
  overview of, 892–896
  quick sorts compared with, 900
  recurrence relations and, 847
  time complexity of, 895
`mergeSort` method, 895
Mersenne prime, 243
`MessagePanel` class
  `ClockPane.java`, 582–584
  `DisplayClock.java`, 581–582
Meters, converting to/from feet, 238
Method header, 207
Method modifiers, 207, 1168–1169
Method reference, 790
Method signature, 207
Methods
  abstraction and, 227–234
  accessing object methods, 333
  calling, 208–210
  case study: converting decimals to hexadecimals, 184–486
  case study: converting a hexadecimal digit to a decimal value, 143–144
  case study: generating random numbers, 225–227
  case study: generic method for sorting array, 758–759
  class, 339–344
  `Collection` interface, 778
  commenting, 18
  `Comparator` interface, 787–789
  defining, 206–208
  generic, 756–758
  identifiers, 40
  implementation methods, 231–234
  invoking, 208, 208, 333, 757
  key terms, 234
  modularizing code, 217–219
  naming conventions, 44
  object actions defined by, 324–326
  overloading, 221–224
  overriding, 1000–1001
  overview of, 206
  passing arrays to, 261–264
  passing objects to, 349–353
  passing arguments by values, 213–217
  passing to two-dimensional arrays, 295–296
  quiz and exercises, 236–248
  recursive methods, 720
  returning arrays from, 264–265
  rounding, 123–124
  static. *see* Static methods
  stepwise refinement, 227–234
  summary, 235
  top-down and/or bottom-up implementation, 229–231
  top-down design, 228–230
  tracing or stepping over as debugging technique, 108
  trigonometric, 123
  variable scope and, 224–225
  `void` method example, 211–213
MHz (Megahertz), clock speed, 3
Microsoft Windows, 9
Million bits per second (Mbps), 7
`min` method
  finding minimum element in lists, 794
  `Math` class, 24
Minimum spanning trees (MSTs)
  MST algorithm, 1103–1105
  overview of, 1092
  Prim's minimum spanning tree algorithm, 1103–1105
  `TestMinimumSpanningTree.java`, 1107–1109
  weighted graphs and, 1087
  `WeightedGraph` class, 1095
Mnemonics, in assembly language, 7
Modeling, graphs and, 1054–1064
Modems (modulator/demodulator), 6
Modifier keys, on keyboards, 6
Modifiers
  list of, 1168–1169
  method modifiers, 207
Modularizing code
  `GreatestCommonDivisorMethod.java`, 217–218
  overview of, 217
  `PrimeNumberMethod.java`, 218–219
Monitors (displays), 6
Motherboard, 3
Mouse, as I/O device, 6
  `ControlCircleWithMouseAndKey.java`, 615–616
  event-driven programming, 611–612
  `MouseEvent`, 611–612
`MouseEvent`, 611–612
MST algorithm, 1103–1105
`MST` class, 1106–1107
MSTs. *see* Minimum spanning trees (MSTs)
Multi-dimensional arrays. *see* Arrays, multi-dimensional
Multimedia. *see* JavaFX UI controls
Multiple-choice test, 296–298
Multiplication (`*=`) assignment operator, 56
Multiplication operator (`*`), 14, 46, 53
Multiplication table, 178, 179
Multiplicities, in object composition, 375
Multiprocessing, 10
Multiprogramming, 10
Multithreading, 10
Multi-way `if-else` statements
  in computing taxes, 92–95
  overview of, 83–85
Mutator methods. *see* Setter (mutator) methods

N

Named constants. *see* Constants
Naming conventions
  class design guidelines, 532
  interfaces, 524
  programming and, 44
  wrapper classes, 382
Naming rules, identifiers, 40
`NavigableMap` interface, 831
N-by-*n* matrix, 241
Negative angles, drawing arcs, 577
Neighbors
  depth-first searches (DFS), 1068
  vertices, 1048
Nested classes. *see* Inner (nested) classes
Nested `if` statements
  in computing body mass index, 91–92
  in computing taxes, 92–95
  overview of, 83
Nested loops, 178–180, 293, 843
NetBeans
  built in debugging, 106
  creating/editing Java source code, 15
Network interface cards (NICs), 6
`new` operator
  creating arrays, 250
  creating objects, 331
`next` method, whitespace characters and, 134
`nextLine()` method, whitespace characters and, 134
Next-line style, block styles, 19
NICs (network interface cards), 6
Nine tails problem
  graphic approach to, 1077–1121
  reducing to shortest path problem, 1114–1117
No-arg constructors
  class design guidelines, 533
  `Loan` class, 370
  wrapper classes not having, 383
Node, 545
Nodes, AVL trees
  balancing on a path, 1000–1001
  creating, 1003
  creating and storing in `AVLTreeNode` class, 999–1000
  deleting elements, 1002
  rotation, 1003–1004
Nodes, binary trees
  deleting leaf node, 975
  overview of, 960
  representing binary search trees, 961–962
Nodes, `JavaFX`, 551–552
Nodes, linked lists
  creating, 940, 941
  deleting, 941–943
  overview of, 935–937
  storing elements in, 939, 940
`noneMatch` method, 1134
Nonleaves, finding, 991
Not (`!`) logical operator, 95–99
Not equal to (`!=`) comparison operator, 78
`NotSerializableException`, 709
`null` values, objects, 334
`NullPointerException`, as runtime error, 334
`Number` class
  case study: abstract number class, 505
  as root class for numeric wrapper classes, 505
Numbers/numeric types
  abstract number class, 505–507
  binary. *see* Binary numbers
  case study: converting hexadecimals to decimals, 143–144, 219–221
  case study: displaying prime numbers, 191–193
  case study: generating random numbers, 225–227
  case study: guessing numbers, 163–166
  casting to/from `char` types, 128–129
  conversion between strings and, 138–140
  converting to/from strings, 392
  decimal. *see* Decimal numbers
  double. *see* `double` (double precision), numeric types
  floating-point. *see* Floating-point numbers (`float` data type)
  generating random numbers, 89–90
  `GreatestCommonDivisorMethod.java`, 217–218
  hexadecimal. *see* Hexadecimal numbers
  integers. *see* Integers (`int` data type)
  `LargestNumbers.java`, 506–507
  overview of, 45–48
  `PrimeNumberMethod.java`, 218–219
  processing large numbers, 387–388
  types of number systems, 1171–1174
Numerators, in rational numbers, 526
Numeric keypads, on keyboards, 6
Numeric literals, 48–50
Numeric operators
  applied to characters, 129
  overview of, 46–47

O

`Object` class, 424, 433–434
`Object I/O`. *see* `ObjectInputStream`/`ObjectOutputStream` classes
Object member access operator (`.`), 333, 431
Object reference variables, 333
`ObjectInputStream`/`ObjectOutputStream` classes
  overview of, 706–707
  `serializable` interface, 708–709
  serializing arrays, 709–711
  `TestObjectInputStream.java`, 708
  `TestObjectOutputStream.java`, 707–708
Object-oriented programming (OOP), 324, 333, 372–375
Objects
  accessing data and methods of, 333
  accessing via reference variables, 332–333
  array of, 263
  `ArrayList` class, 434–435
  arrays of, 353–355
  automatic conversion between primitive types and wrapper class types, 386
  `BigInteger` and `BigDecimal` classes, 387–388
  cannot be created from abstract classes, 504
  case study: designing class for stacks, 380–382
  case study: designing `Course` class, 378–380
  casting, 429–433
  in `Circle.java` (for `CircleWithPrivateDataFields`) example, 347–348
  `Circle.java` (for `CircleWithStaticMembers`) example, 340–341
  class abstraction and encapsulation, 368–372
  class design guidelines, 531–534
  classes from Java Library, 336–339
  comparing primitive variables with reference variables, 334–336
  composing, 376–377
  constructors, 331
  creating, 326–331
  data field encapsulation for maintaining classes, 346–349
  `Date` class, 336–337
  defining classes for, 324–326

- Objects (*Continued*)
  - edges defined as, 1052
  - `equals` method of `Object` class, 433–434
  - event listener object, 597
  - event objects, 596
  - immutable, 355–356
  - inheritance. *see* Inheritance
  - key terms, 361, 401
  - `Loan.java`, 370–372
  - `null` values, 334
  - `Object` class, 424
  - object-oriented thinking, 372–375
  - `ObservablePropertyDemo.java`, 616–617
  - overview of, 324, 368
  - passing to methods, 349–353
  - processing primitive data type values as, 382–386
  - quiz and exercises, 362–366, 401–410
  - `Random` class, 337–338
  - reference data fields and, 334
  - representing edges, 1051
  - `ResizableCircleRectangle.java`, 617–618
  - static variables, constants, and methods and, 339–344
  - summary, 361–362, 401
  - `TestCircle.java` example, 326–327
  - in `TestCircleWithPrivateDataFields.java` example, 348–349
  - in `TestCircleWithStaticMembers.java` example, 341–344
  - in `TestTV.java` example, 330–331
  - `this` reference and, 358–361
  - `TotalArea.java` example, 354–355
  - in `TV.java` example, 329–330
  - variable scope and, 357–358
  - vertices as object of any type, 1049
  - visibility modifiers, 344–348
- Off-by-one errors
  - arrays and, 255
  - in loops, 162
- OOP (Object-oriented programming), 324, 333, 372–375
- Open addressing, hashing
  - collision handling using, 1019–1023
  - double hashing, 1022–1023
  - linear probing, 1019–1020
  - quadratic probing, 1020–1021
- Operands
  - defined, 97
  - incompatible, 97
- Operators
  - assignment operator (`=`), 42–43
  - augmented assignment operators, 56–57
  - bit operators, 1175
  - comparison operators, 78
  - increment and decrement operators, 57–58
  - numeric operators, 46–48
  - precedence and associativity, 106–107
  - precedence and associativity chart, 1166–1167
  - processing, 804
  - unary and binary, 48
- Option buttons. *see* Radio buttons
- Or (`||`) logical operator, 95–99
- OSs (operating systems)
  - overview of, 9
  - tasks of, 10
- Output
  - redirection, 170
  - streams, 692
- `OutputStream` classes
  - `BufferedOutputStream`, 701–704
  - case study: copying files, 705
  - `DataOutputStream`, 698–701
  - `DetectEndOfFile.java`, 701
  - `FileOutputStream`, 695–696
  - `FilterOutputStream`, 698
  - `ObjectOutputStream`, 706–707
  - overview of, 694–695
  - serialization and, 709
  - `TestDataStream.java`, 699–700
  - `TestFileStream.java`, 696–698
  - `TestObjectOutputStream.java`, 707–708
- Overflows
  - `Rational` class, 530
  - variables, 66
- Overloading methods, 221–224
- Overriding methods, 421–422, 1000–1001

P

- π (pi), estimating, 240
- Package-private (package-access) visibility modifiers, 344
- Packages
  - organizing classes in, 345
  - organizing programs in, 18
- *Page Down key,* on keyboards, 6
- *Page Up key,* on keyboards, 6
- Pair of points, algorithm for finding closest, 861–864
- Palindromes
  - case study: checking if string is a palindrome, 189–191
  - case study: ignoring nonalphanumeric characters when checking palindromes, 398–400
  - palindrome integers, 236
  - palindromic primes, 243
  - `RecursivePalindrome.java`, 728–729
  - `RecursivePalindromeUsingSubstring.java`, 727–728
- Panels
  - `ButtonInPane.java`, 546
  - `MessagePanel` class. *see* `MessagePanel` class
- Parallel edges, 1048
- Parallel execution, order of, 1140
- Parallel streams
  - overview of, 1139
  - `ParallelStreamDemo.java` example, 1139–1141
  - *vs.* sequential streams, 1140–1141
- Parameters
  - actual parameters, 207
  - defining methods and, 206–207
  - generic classes, 756
  - generic methods, 758
  - generic parameters not allowed in static context, 765–766
  - as local variable, 224
  - order association, 214
  - passing by values, 214–217
  - variable-length argument lists, 268–269
- Parent, 545
- Parentheses (`( )`)
  - defining and invoking methods and, 227
  - in `Welcome.java`, 14
- Parsing methods, 384
- Pascal, high-level languages, 8
- Pass-by-sharing
  - arrays to methods, 262
  - objects to methods, 350
- Pass-by-value
  - arrays to methods, 262
  - `Increment.java` example, 214

  objects to methods, 349
  overview of, 214
  `TestPassByValue.java` example, 215–217
`PasswordField`, 655
Passwords, checking if string is valid password, 241
`PathTransition`, 619–622
Pentagonal numbers, 236
Perfect hash function, 1016
Perfectly balanced trees, 996
Pivot element, 896
Pixels (picture elements), measuring resolution in, 6
Points
  algorithm for finding closest pair of, 861–864
  finding convex hull for a set of points, 867–869
`Polygon` and `Polyline`
  overview, 577
  `ShowPolygon.java`, 578–580
Polymorphism
  `CastingDemo.java` example, 430–431
  overview of, 425
  `PolymorphismDemo.java` example, 425
Polynomial hash codes, 1018
Postfix decrement operator, 57–58
Postfix increment operator, 57–58
Postfix notation, 812–813
Postorder traversal
  time complexity of, 978
  tree traversal, 963
Posttest loops, 176
`pow` method, `Math` class, 48
Precedence, operator, 106–107, 1166–1167
Prefix decrement operator, 57–58
Prefix increment operator, 57–58
Preorder traversal
  time complexity of, 978
  tree traversal, 963
Pretest loops, 176
Prime numbers
  algorithm for finding, 855–861
  case study: displaying prime numbers, 191–193
  comparing prime number algorithms, 861
  `EfficientPrimeNumbers.java` example, 857–859
  `PrimeNumberMethod.java`, 218–219
  `PrimeNumbers.java` example, 856–857
  `SieveOfEratosthenes.java` example, 860–861
  types of, 243
Primitive types (fundamental types)
  automatic conversion between primitive types and wrapper class types, 386, 753
  casting, 431
  comparing parameters of primitive type with parameters of reference types, 351
  comparing primitive variables with reference variables, 334–336
  converting wrapper object to/from (boxing/unboxing), 386
  creating arrays of, 353
  hash codes for, 1017
Prim's minimum spanning tree algorithm
  Dijkstra's algorithm compared to, 1105
  overview of, 1097–1106
`print` method, `PrintWriter` class, 38, 480–481
`printf` method, `PrintWriter` class, 480–481
Printing arrays, 293
`println` method, `PrintWriter` class, 38, 480–481
`printStackTrace` method, 465–466
`PrintWriter` class
  case study: replacing text, 485–486
  writing data to file using, 480–481
  for writing text data, 692
Priority queues
  implementing, 953–954
  `MyPriorityQueue.java` example, 953–954
  overview of, 800
  `PriorityQueue` class, 802
  for storing weighted edges, 1102
  `TestPriorityQueue.java` example, 954
`PriorityQueue` class, 802
`private`
  encapsulation of data fields and, 346–347
  visibility modifier, 345–346, 443–444
Problems
  breaking into subproblems, 192
  creating programs to address, 34
  solving with recursion, 726–728
Procedural paradigm, compared with object-oriented paradigm, 374–375
Procedures, 207. *see also* Methods
Processing arrays, 253–255
Processor, 1147
Programming errors. *see also* Exception handling
  `ClassCastException`, 430
  debugging, 108
  logic errors, 21
  minimizing numeric errors related to loops, 180–181
  runtime errors, 20
  selections, 85–89
  syntax errors, 13, 19
  using generic classes for detecting, 752–754
Programming languages
  assembly language, 7
  high-level languages, 8–9
  Java. *see* Java programming
  machine language, 7
  overview of, 2
Programming style
  block styles, 19
  comments and, 18–19
  indentation and spacing, 19
  overview of, 18
Programs/programming
  assignment statements and expressions, 42–43
  case study: counting monetary units, 64–67
  case study: displaying current time, 54–55
  character data type, 126–131
  coding incrementally, 164
  evaluating expressions and operator precedence rules, 52–54
  exponent operations, 48
  identifiers, 40
  increment and decrement operators, 57–58
  introduction to, 34
  with Java language. *see* Java programming
  key terms, 68
  modularizing code, 217–219
  named constants, 43–44
  naming conventions, 44
  numeric literals, 48–53
  numeric operators, 46–48
  numeric type conversions, 58–60
  numeric types, 45
  overview of, 2
  questions and exercises, 70–75
  reading input from console, 37–39
  recursive methods in, 720
  software development process, 61–64

Programs/programming (*Continued*)
  string data type, 131–140
  summary, 69–70
  variables, 40–42
  writing a simple program, 34–37
protected
  data and methods, 442–444
  visibility modifier, 345, 443–444
Protected data fields, 999
Pseudocode, 34
Public classes, 327
public method, 348
public visibility modifier, 344–346, 442–444
Python, high-level languages, 8

Q

Quadratic algorithm, 843, 848
Quadratic probing, collision handling, 1020–1021
Query methods, **Map** interface, 829
Query operations, Collection interface, 777
Queue interface, 800
Queues
  breadth-first search algorithm, 1074
  bucket sorts and, 908–909
  as collection type, 776
  Dequeue interface, 800–802
  GenericQueue.java example, 951
  implementing, 949–953
  overview of, 800
  priority queues. *see* Priority queues
  Queue interface, 800
  TestStackQueue.java example, 951–953
  WeightedGraph class, 1095–1102
Quick sorts
  algorithm, 895–897
  merge sorts compared with, 900
  overview of, 895
  QuickSort.java example, 897–900
Quincunx, 288
Quotients
  Quotient.java example, 454
  QuotientWithException.java example, 456–458
  QuotientWithIf.java example, 455
  QuotientWithMethod.java example, 455–456

R

Radio buttons, 651–653
RadioButtonDemo.java, 652–653
Radix sorts, 907–909
Ragged arrays, 292–293, 1052
RAM (random-access memory), 4
Random class, **java.util**, 337
random method
  case study: generating random numbers, 225–227
  case study: lottery, 100–101
  Math class, 89–90, 124
Random numbers
  case study: generating random numbers, 225–227
  case study: lottery, 100–101
  generating, 89–90
Random-access files
  overview of, 711–712
  TestRandomAccessFile.java, 713–714
Random-access memory (RAM), 4
Rational class
  case study: designing class for matrix using generic types, 766–767
  overview of, 526–527
  Rational.java example, 528–531
  RationalMatrix.java example, 769–770
  TestRationalClass.java example, 527–528
  TestRationalMatrix.java example, 770–771
Rational numbers, representing and processing, 526–528
Raw types, backward compatiblity and, 760–761
readASolution() method, applying to Sudoku grid, 302
Read-only streams, 711. *see also* **InputStream** classes
Read-only views, **Collections** class, 835
Rebalancing AVL trees, 996–998
Rectangle
  overview, 570–571
  ShowRectangle.java, 571–572
Recurrence relations, in analysis of algorithm complexity, 847
Recursion
  binary searches, 730
  case study: computing factorials, 720–723
  case study: computing Fibonacci numbers, 723–726
  case study: determining directory size, 731–732
  case study: fractals, 736–739
  case study: Towers of Hanoi, 33–736
  ComputeFactorial.java, 721–723
  ComputeFactorialTailRecursion.java, 741
  ComputeFibonacci.java, 724–726
  depth-first searches (DFS), 1068
  DirectorySize.java, 731–732
  displaying/visualizing binary trees, 980
  helper method, 728
  iteration compared with, 740
  key terms, 741
  overview of, 720
  problem solving by thinking recursively, 726–728
  questions and exercises, 742–750
  RecursivePalindrome.java, 728–729
  RecursivePalindromeUsingSubstring.java, 727–728
  RecursiveSelectionSort.java, 729
  selection sorts, 729
  SierpinskiTriangle.java, 736–739
  summary, 742
  tail recursion, 740–741
  TowersOfHanoi.java, 734–736
Recursive methods, 720
Red–black trees, 1016
reduce method, stream reduction using, 1141–1144
Reduction, characteristics of recursion, 726
Reference data fields, 359
Reference types
  classes as, 332
  comparing parameters of primitive type with parameters of reference types, 351
  comparing primitive variables with, 334–336
  generic types as, 752
  reference data fields, 334
  string data type as, 131
Reference variables
  accessing objects with, 332–333
  array of objects as array of, 353
  comparing primitive variables with, 334–336
Register listeners
  ControlCircle.java, 600–601
  ControlCircleWithMouseAndKey.java, 598–599, 615–616
  KeyEventDemo.java, 614–615
  LoanCalculator.java, 610–611
  overview of, 597–598

- Regular expressions
  - matching strings with, 391, 1176
  - replacing and splitting strings, 1179–1180
  - syntax, 1176–1179
- Rehashing
  - load factor and, 1025
  - time complexity of hashing methods and, 1033
- Relative file names, 477–478
- Remainder (`%=`) assignment operator, 56–57
- Remainder (`%`) or modulo operator, 46, 52
- `remove` method, linked lists, 935, 938
- Repetition
  - determining Big *O* for repetition statements, 842–845
  - loops. *see* Loops
- `replace` method, strings, 390
- `replaceAll` method, strings, 390, 1179
- `replaceFirst` method, strings, 390, 1179
- Requirements specification, in software development process, 61
- Reserved words. *see* Keywords (reserved words)
- Resources, role of OSs in allocating, 10
- Responsibilities, separation as class design principle, 532
- `return` statements, 209
- Return value type
  - constructors not having, 331
  - in defining methods, 207
- Reusable code
  - benefits of stepwise refinement, 234
  - code modularization and, 217
  - method enabling, 210
  - methods for, 206
- `reverse` method
  - applying to lists, 792
  - returning arrays from methods, 262
- Right subtree, of binary trees, 960
- Right-heavy, balancing AVL nodes, 996, 1004
- RL imbalance, AVL nodes, 997, 998
- RL rotation
  - `AVLTree` class, 1004, 1005
  - balancing nodes on a path, 1000
  - options for balancing AVL nodes, 997, 998
- Root, of binary trees, 960, 961
- Rotation
  - `AVLTree` class, 1003–1004
  - balancing nodes on a path, 1000–1001
  - implementing, 1001
  - methods for performing, 1007
  - options for balancing AVL nodes, 996–998
- Rounding methods, `Math` class, 123–124
- Row index, 291
- RR imbalance, AVL nodes, 996, 998
- RR rotation
  - `AVLTree` class, 1004, 1005
  - balancing nodes on a path, 1000
  - options for balancing AVL nodes, 996, 998
- Runtime errors
  - debugging, 108
  - declaring, 461
  - exception handling and, 39, 454
  - `NullPointerException` as, 334
  - programming errors, 21
- Runtime stacks. *see* Call stacks

S

- `Scanner` class
  - obtaining input with, 68
  - for reading console input, 37–39
  - reading data from file using, 82–483
  - for reading text data, 692
- Scanners
  - case study: replacing text, 485–486
  - creating, 458
- `Scene`, 542–545
- Scheduling operations, 10
- Scientific notation, of floating-point literals, 50
- Scope, of variables, 41, 224–225
- Screen resolution, 6
- Scroll bars
  - overview of, 665–666
  - `ScrollBarDemo.java`, 666–667
- Scroll panes
  - `DescriptionPanel.java`, 657
  - overview of, 656
  - scrolling lists, 663
- `search` method, `AVLTree` class, 1011
- Searches
  - arrays, 267
  - binary search trees. *see* Binary search trees (BST)
  - binary searches, 270–273, 730
  - linear searches, 269–270
  - recursive approach to searching for words, 720
  - search keys, 1016, 1042
- `SearchTree` class
  - as inner class of `UnweightedGraph` class, 1062
  - `MST` class extending, 1101–1102
  - `ShortestPathTree` class extending, 1101
  - traversing graphs and, 1067
- Secondary clustering, quadratic probing issue, 1021
- Segments, merging, 913
- Selection sort algorithm
  - analyzing, 846
  - recurrence relations and, 847
- Selection sorts
  - arrays, 273–274
  - `RecursiveSelectionSort.java`, 729
  - using recursion, 729
- Selection statements, 78, 80
  - determining Big *O* for, 842–845
- Selections
  - `Addition.Quiz.java` example, 79–80
  - `boolean` data type, 78–79
  - case study: computing Body Mass Index, 91–92
  - case study: computing taxes, 92–93
  - case study: determining leap year, 99–100
  - case study: guessing birthdays, 140–143
  - case study: lottery, 100–101
  - common errors, 85–89
  - conditional operators, 105–106
  - debugging, 108
  - formatting output consoles, 146–150
  - generating random numbers, 89–90
  - `if` statements, 80–81
  - `if-else` statements, 82–83
  - key terms, 109
  - logical operators, 95–99
  - nested `if` statements and multi-way `if-else` statements, 83–85
  - operator precedence and associativity, 106–107
  - overview of, 78
  - questions and exercises, 110–120
  - summary, 109
  - `switch` statements, 102–105
- Semicolons (`;`), common errors, 86
- Sentinel-controlled loops, 168–170

Separate chaining
  handling collision in hashing, 1023
  implementing map using hashing, 1025
Sequence statements, determining Big *O* for, 842–845
Sequential files, input/output streams, 711
Sequential streams, 1139
  parallel streams *vs.*, 1140–1141
Serialization
  of arrays, 709–711
  of objects, 709
`set` method, `List` interface, 783
Set operations, `Collection` interface, 777
`setLength` method, `StringBuilder` class, 397
`setRadius` method
  `Circle` example, 327
  `CircleWithPrivateDataFields.java` example, 348
Sets
  case study: counting keywords, 827–828
  as collection type, 776
  comparing list performance with, 824–826
  `HashSet` class, 816–820
  key terms, 836
  `LinkedHashSet` class, 820
  overview of, 816
  quiz and exercises, 836–838
  singleton and unmodifiable, 835
  summary, 836
  `TestHashSet.java` example, 817–818
  `TestLinkedHashSet.java` example, 820
  `TestMethodsInCollection.java` example, 818–819
  `TestTreeSet.java` example, 821
  `TestTreeSetWithComparator.java` example, 823–824
  `TreeSet` class, 820–824
Sets, implementing with hashing
  `MyHashSet.java` example, 1034–1041
  overview of, 1034
  `TestMyHashSet.java` example, 1041–1042
Setter (mutator) methods
  `ArrayList` class and, 438
  encapsulation of data fields and, 347–348
  implementing linked lists, 935
Seven Bridges of Königsberg problem, 1047
Shallow copies, `clone` method and, 520–521
Shapes, 545–548
  `Arc`, 575–577
  `Circle` and `Ellipse`, 572–574
  `Line`, 569–570
  `Polygon` and `Polyline`, 577–580
  `Rectangle`, 570–572
  `Text`, 567–569
Sharing code, 210
`short`, numeric types
  hash codes for primitive types, 1017
  overview of, 45
Short-circuit operator, 98
Shortest path tree, 1109
Shortest paths
  case study: weighted nine tails problem, 1118–1119
  Dijkstra's algorithm, 1110–1111
  finding with graph, 1046, 1049
  nine tails problem, 1077–1083
  overview of, 1105–1106
  `TestShortestPath.java`, 1115–1116
  `WeightedGraph` class and, 1095
`ShortestPathTree` class, 1110
Shuffling arrays, 254, 294–295
Sibling, 960
Sierpinski triangle
  case study, 736–737
  computing recursively, 744, 747, 749
  `SierpinskiTriangle.java`, 736–739
Sieve of Eratosthenes, 859–861
Simple graphs, 1048
`sin` method, trigonometry, 122–123
Single abstract method (SAM) interface, 607
Single precision numbers. *see* Floating-point numbers (`float` data type)
Single-dimensional arrays. *see* Arrays, single-dimensional
Single-source shortest-path algorithm, Dijkstra's, 1110–1111
Singly linked lists. *see* `LinkedList` class
Sinking sorts, 283, 890–892
Sliders
  overview of, 668
  `SliderDemo.java`, 669–670
Software
  development process, 61–64
  programs as, 2
`sort` method
  `Arrays` class, 275, 276
  `ComparableRectangle.java` example, 515–516
  lists and, 792
  `SortRectangles.java` example, 516–517
  using recursion, 730
`sorted` method, 1133–1134
`SortedMap` interface, 830, 831
Sorting
  adding nodes to heaps, 901
  arrays using heaps, 906
  bubble sort, 890–892
  bucket sorts and radix sorts, 907–909
  complexity of external sorts, 916
  complexity of heap sorts, 906–907
  `CreateLargeFile.java` example of external sorts, 909–910
  external sorts, 909–910
  `Heap` class and, 904–905
  heap sort, 900–907
  `Heap.java` example, 904–905
  `HeapSort.java` example, 906
  implementation phases of external sorts, 911–915
  insertion sorts, 888–890
  key terms, 916
  merge sorts, 892–895
  overview of, 888
  quick sort, 896–900
  quiz and exercises, 917–921
  removing root from heap, 902–903
  storing heaps, 901
  summary, 917
Sorting arrays
  bubble sorts, 283
  case study: generic method for, 758–759
  insertion sorts, 888–892
  overview of, 273
  selection sorts, 273–274
Source objects, event sources and, 596–597
Source program or source code, 8
Space complexity, 841
Spacing, programming style and, 19
Spanning trees
  graphs, 1048
  minimum spanning trees, 1099–1101
  MST algorithm, 1103–1105
  Prim's minimum spanning tree algorithm, 1103–1105

  - `TestMinimumSpanningTree.java`, 1107–1108
  - traversing graphs and, 1067
- Special characters, 13
- Specific import, 38
- `split` method, strings, 390, 391, 1179, 1180
- `Stack` class, 799
- `StackOfIntegers` class, 380–381
- `StackOverflowError`, recursion causing, 740
- Stacks
  - case study: designing class for stacks, 380–382
  - case study: evaluating expressions, 803–804
  - `EvaluateExpression.java` example, 805–808
  - `GenericStack` class, 755–756
  - implementing, 949–953
  - `Stack` class, 799
  - `TestStackQueue.java` example, 951–953
- `Stage`, 542, 545
- State, of objects, 324
- Statements
  - `break` statements, 103
  - `continue` statements, 187–188
  - executing one at a time, 108
  - executing repeatedly (loops), 160
  - in high-level languages, 8
  - `if`. *see* `if` statements
  - `if-else`. *see* `if-else` statements
  - `return` statements, 209
  - `switch` statements, 102–103
  - terminators, 12
- Static methods
  - in `Circle.java` (for `CircleWithStaticMembers`), 340–341
  - class design guidelines, 533
  - declaring, 340
  - defined, 340
  - for lists and collections, 792–795
  - `Stream` interface, 1130
  - when to use instance methods *vs.* static, 341–344
  - wrapper classes and, 384
- Static variables
  - in `Circle.java` (for `CircleWithStaticMembers`), 340–341
  - class, 339–344
  - class design guidelines, 533
  - declaring, 340
  - instance variables compared with, 339–341
  - in `TestCircleWithStaticMembers.java` example, 341
  - when to use instance variables *vs.* static, 341–343
- Stepwise refinement
  - benefits, 234
  - implementation methods, 231–234
  - method abstraction, 227–234
  - top-down and/or bottom-up implementation, 229–231
  - top-down design, 228–231
- Storage devices
  - CDs and DVDs, 5
  - Cloud storage, 5
  - disks, 5
  - overview of, 4–5
  - USB flash drives, 5
- Storage units, for measuring memory, 3
- `Stream.of` method, 1133
- Streams, 1130
  - `AnalyzeNumbersUsingStream.java` example, 1150–1151
  - case study: analyzing numbers, 1150–1152
  - case study: counting keywords, 1155–1156
  - case study: counting occurrences of each letter, 1151–1152
  - case study: counting occurrences of each letter in string, 1152–1153
  - case study: finding directory size, 1154–1155
  - case study: occurrences of words, 1157–1158
  - case study: processing all elements in two-dimensional array, 1153–1154
  - `CollectDemo.java` example, 1145–1147
  - `CollectGroupDemo.java` example, 1148–1150
  - `CountKeywordStream.java` example, 1155–1156
  - `CountLettersUsingStream.java` example, 1151–1152
  - `CountOccurrenceOfLettersInAString.java` example, 1152–1153
  - `CountOccurrenceOfWordsStream.java` example, 1157–1158
  - `DirectorySizeStream.java` example, 1154–1155
  - `DoubleStream`, 1136–1139
  - grouping elements using `groupingby` collector, 1147–1150
  - `IntStream`, 1136–1139
  - `IntStreamDemo.java` example, 1136–1139
  - `LongStream`, 1136–1139
  - overview of, 1130
  - parallel streams, 1139–1141
  - `ParallelStreamDemo.java` example, 1139–1141
  - quiz and exercises, 1158–1159
  - `Stream` class, 1131
  - stream pipelines, 1130–1136
  - stream reduction using `collect` method, 1144–1147
  - stream reduction using `reduce` method, 1141–1144
  - `StreamDemo.java` example, 1132–1133
  - `StreamReductionDemo.java` example, 1142–1144
  - summary, 1158
  - `TwoDimensionalArrayStream.java` example, 1153–1154
- `String` class, 388
- String concatenation operator (`+`), 36
- String literals, 388
- String matching
  - `StringMatch.java`, 870
  - Boyer-Moore Algorithm, 870–873
- String variables, 389
- `StringBuffer` class, 395–396
- `StringBuilder` class
  - case study: ignoring nonalphanumeric characters when checking palindromes, 398–400
  - modifying strings in, 395–397
  - overview of, 395
  - `toString, capacity, length, setLength`, and `charAt` methods, 397
- Strings
  - in binary I/O, 698–699
  - case study: checking if string is a palindrome, 189–191
  - case study: converting hexadecimals to decimals, 219–221
  - case study: counting the occurrences of each letter in a string, 1152–1153
  - case study: ignoring nonalphanumeric characters when checking palindromes, 398–401
  - case study: revising the lottery program, 145–146
  - `Character` class, 189–191
  - command-line arguments, 276–279
  - concatenating, 36, 131
  - constructing, 388
  - conversion between numbers and, 138–139
  - converting to/from arrays, 392
  - finding characters or substrings in, 390–391
  - formatting, 392–395
  - generic method for sorting array of `Comparable` objects, 758–759
  - hash codes for, 1017–1018
  - immutable and interned, 388–390
  - matching, replacing, and splitting by patterns, 391, 1179–1180
  - overview of, 388
  - replacing, and splitting, 390
  - `string` data type, 131
  - `StringBuilder` and `StringBuffer` classes, 395–401
  - substrings, 37, 137
  - in `Welcome.java`, 12

Subclasses
  abstract methods and, 501
  abstracting, 504
  constructors, 418
  of Exception class, 460–461
  inheritance and, 412–418
  of `RuntimeException` class, 461
Subdirectories, 731
Subgraphs, 1048
Subinterfaces, 523
`substring` method, 136, 137, 728
Substrings, 137
Subtraction (`-=`) assignment operator, 56–57
Subtraction (`-`) operator, 46, 53
Subtrees
  of binary trees basics, 960
  searching for elements in BST, 962
Sudoku puzzle, 300–303, 884–885
`sum` method, 206, 1137, 1138
`super` keyword, 418
Superclass methods, 420–421
Superclasses
  of abstract class can be concrete, 504
  classes extending, 523
  inheritance and, 412–418
  subclasses related to, 501
Supplementary characters, Unicode, 127
`swap` method
  swapping elements in an array, 263–264
  in `TestPassByValue.java` example, 215, 216
`switch` statements
  `ChineseZodiac.java` example, 104–105
  with enumerated types, 1183
  overview of, 102–103
Syntax errors (compile errors)
  common errors, 13, 14
  debugging, 108
  programming errors, 19
Syntax rules, in `Welcome.java`, 13–14
System activities, role of OSs, 10
System analysis, in software development process, 61–63
System design, in software development process, 61, 63
System errors, 460
`System.in`, 37
`System.out`, 37, 146–150

T

Tables, storing, 290
Tail recursion
  `ComputeFactorialTailRecusion.java`, 741
  overview of, 740–741
`tan` method, trigonometry, 122–123
TBs (terabytes), of storage, 4–5
Teamwork, facilitated by stepwise refinement, 234
Terabytes (TBs), of storage, 4–5
Terminal method, `Stream` interface, 1130
Testing
  benefits of stepwise refinement, 234
  in software development process, 62, 64
`TestShortestPath.java`, 1115–1116
Text
  case study: replacing text, 485–487
  files, 692
  overview, 567
  `ShowText.java`, 568–569
  `TextArea`, 655–658
  `TextAreaDemo.java`, 658
  `TextField`, 654–655
  `TextFieldDemo.java`, 654–655
  .txt files (text), 694
Text I/O
  vs. binary I/O, 693–694
  handling in Java, 692–693
  overview of, 692
TextPad, for creating/editing Java source code, 12
`thenComparing` method, 791
`this` reference
  invoking constructors with, 360
  overview of, 358
  referencing data fields with, 359
Three-dimensional arrays. *see* Arrays, multi-dimensional
`throw` keyword
  chained exceptions, 473–474
  `throw ex` for rethrowing exceptions, 473
  for throwing exceptions, 467
`Throwable` class
  generic classes not extending, 766
  getting information about exceptions, 465
  `java.lang`, 459–461
Throwing exceptions, 462–463, 467–470
  `CircleWithException.java` example, 467
  `QuotientWithException.java` example, 456–458
  rethrowing, 472–473
  `TestCircleWithCustomException.java` example, 476
  `throw` keyword for, 463
`throws` keyword
  chained exceptions, 473–474
  for declaring exceptions, 462, 467
  `IOException`, 695, 696
  for throwing exceptions, 463
Tic-tac-toe game, 310
Time complexity, 841
  AVL trees, 1011
  `BST` class, 978
  bubble sort, 892
  heap sorts, 906–907
  insertion sorts, 890
  merge sorts, 895
  rehashing, 1033
`toArray` method, 1132, 1135–1136
`toCharArray` method, converting strings into arrays, 392
`ToggleButton`, 651
`ToggleGroup`, 652
Token reading methods, `Scanner` class, 484–485
Top-down design, 228–231
Top-down implementation, 229–231
`toString` method
  `ArrayList` class, 437
  `Arrays` class, 276
  `Date` class, 337
  implementing `MyLinkedList`, 947
  `MyArrayList.java` example, 932, 933
  `Object` class, 433
  `StringBuilder` class, 397
`total` variable, for storing sums, 294
Touchscreens, 6
Towers of Hanoi problem
  analyzing algorithm for, 846–847
  computing recursively, 744
  nonrecursive computation, 814
  recurrence relations and, 847

Tracing a program, 36
`transient` keyword, serialization and, 709
Transistors, CPUs, 3
Traveling salesperson problem (TSP), 1123
Traversing binary search trees, 963–965
Traversing graphs
  breadth-first searches (BFS), 1074–1077
  case study: connected circles problem, 1072–1076
  depth-first searches (DFS), 1068–1072
  overview of, 1068
  `TestWeightedGraph.java`, 1100
`Tree` interface, `BST` class, 965
Tree traversal, 963–965
`TreeMap` class
  case study: counting occurrence of words, 833–834
  concrete implementation of `Map` class, 828–830
  implementation of `Map` class, 1016
  overview of, 831
  `TestMap.java` example, 831–833
  types of maps, 828–829
Trees
  AVL trees. *see* AVL trees
  binary search. *see* Binary search trees (BST)
  connected graphs, 1048
  creating BFS trees, 1075
  Huffman coding. *see* Huffman coding trees
  overview of, 960
  Red–black trees, 1016
  spanning trees. *see* Spanning trees
  traversing, 963–965
`TreeSet` class
  implementation of `Set` class, 1034
  overview of, 820–821
  `TestTreeSet.java` class, 821
  `TestTreeSetWithComparator.java` class, 821–823
  `TestTreeSetWithComparator.java` example, 508–510
  types of sets, 816
Trigonometric methods, `Math` class, 122–123
`trimToSize` method, 932
True/false (Boolean) values, 78
Truth tables, 95–96
`try-catch` blocks
  catching exceptions, 461, 463–465
  chained exceptions, 473–474
  exception classes cannot be generic, 766
  `InputMismatchExceptionDemo.java` example, 458–459
  `QuotientWithMethod.java` example, 455–456
  rethrowing exceptions, 472–473
  `TestCircleWithException.java` example, 468–470
  when to use exceptions, 472–473
Twin primes, 244
Two-dimensional arrays. *see* Arrays, two-dimensional
Type casting
  between `char` and numeric types, 128
  generic types and, 754
  loss of precision, 66
  for numeric type conversion, 58–59
Type erasure, erasing generic types, 764–765

U

UML (Unified Modeling Language)
  aggregation shown in, 376
  class diagrams with, 325
  diagram for `Loan` class, 369
  diagram of `StackOfIntegers`, 380
  diagram of static variables and methods, 339–340
Unary operators, 48
Unbounded wildcards, 762
Unboxing, 386
Unchecked exceptions, 461
Unconditional AND operator, 98
Underflow, floating point numbers, 67
Undirected graphs, 1047
Unicode
  character data type (`char`) and, 126–130
  data input and output streams, 698–699
  generating random numbers and, 225
  text encoding, 692
  text I/O *vs.* binary I/O, 693–694
Unified Modeling Language. *see* UML (Unified Modeling Language)
Uniform Resource Locators. *see* URLs (Uniform Resource Locators)
Unique addresses, for each byte of memory, 4
Universal serial bus (USB) flash drives, 5
UNIX epoch, 54
Unweighted graphs
  defined, 1048
  modeling graphs and, 1048, 1055
  `UnweightedGraph.java` example, 1058–1064
Upcasting objects, 429
Update methods, `Map` interface, 829
`URL` class, `java.net`, 487
URLs (Uniform Resource Locators)
  `ReadFileFromURL.java` example, 487–488
  reading data from Web, 487–488
USB (universal serial bus) flash drives, 5
UTF, 699. *see also* Unicode

V

`valueOf` methods
  converting strings into arrays, 391
  wrapper classes and, 384
Value-returning methods
  `return` statements required by, 209
  `TestReturnGradeMethod.java`, 211–213
  `void` method and, 207
Values
  hashing functions, 1016
  maps and, 1042
`values` method, `Map` interface, 830
Variable-length argument lists, 268–269
Variables
  Boolean variables. *see* Boolean variables
  comparing primitive variables with reference variables, 334–336
  control variables, in for loops, 173–174
  declaring, 35–36, 41
  declaring array variables, 250
  declaring for two-dimensional arrays, 290–291
  displaying/modifying, 108
  hidden, 357
  identifiers, 40
  naming conventions, 44
  overflow, 67
  overview of, 40–42
  reference variables, 332–333
  scope of, 41, 225–226, 357–358
  static variables, 339–344
`Vector` class
  methods, 798–799
  overview of, 798
  `Stack` class extending, 800
Vertex-weighted graphs, 1093

Vertical scroll bars, 666
Vertical sliders, 668, 669
Vertices
  adjacent and incident, 1048
  depth-first searches (DFS), 1068
  `Graph.java` example, 1056
  on graphs, 1048
  Prim's algorithm and, 1099
  representing on graphs, 1048–1050
  shortest paths. *see* Shortest paths
  `TestBFS.java`, 1076
  `TestGraph.java` example, 1056
  `TestMinimumSpanningTree.java`, 1107
  `TestWeightedGraph.java`, 1100
  vertex-weighted graphs, 1093
  weighted adjacency matrices, 1094
`WeightedGraph` class, 1091–1092
Video, `MediaDemo.java`, 677–679
Virtual machines (VMs), 16. *see also* JVM (Java Virtual Machine)
Visibility modifiers, 1169
Visibility (accessibility) modifiers
  classes and, 344–346
  `protected, public`, and `private`, 442–444
Visual Basic, high-level languages, 8
Visualizing (displaying) graphs
  `Displayable.java` example, 1064
  `DisplayUSMap.java` example, 1066–1067
  `GraphView.java` example, 1064–1065
  overview of, 1064
VLSI (very large-scale integration), 720
VMs (virtual machines), 15. *see also* JVM (Java Virtual Machine)
`void` method
  defined, 207
  defining and invoking, 211
  `TestVoidMethod.java`, 211

W

Web, reading file data from, 487–488
Weighted graphs
  case study: weighted nine tails problem, 840–843
  defined, 1048
  Dijkstra's single-source shortest-path algorithm, 1109–1111
  `getMinimumSpanningTree` method, 1103
  key terms, 843
  minimum spanning trees, 1103
  modeling graphs and, 1054
  MST algorithm, 1103–1104
  overview of, 1046–1047
  Prim's minimum spanning tree algorithm, 1103–1105
  priority adjacency lists, 1103–1104
  questions and exercises, 1112–1128
  representing, 1093
  shortest paths, 1109
  summary, 1122
  `TestMinimumSpanningTree.java`, 1107–1109
  `TestShortestPath.java`, 1115–1118
  `TestWeightedGraph.java`, 1100–1102
  weighted adjacency matrices, 1094
  weighted edges using edge array, 1093–1094
  `WeightedGraph` class, 1095–1096
  `WeightedGraph.java`, 1096–100
`WeightedEdge` class, 1088
`WeightedGraph` class
  `getMinimumSpanningTree` method, 1103–1104
  overview of, 1095
  `ShortestPathTree` class as inner class of, 1114
  `TestWeightedGraph.java`, 1100–1101
  `WeightedGraph.java`, 1096–1100
Well-balanced trees
  AVL trees, 996
  binary search trees, 1016
`while` loops
  case study: guessing numbers, 163–166
  case study: multiple subtraction quiz, 166–168
  case study: predicting future tuition, 183
  deciding when to use, 176–178
  design strategies, 166
  `do-while` loops. *see* `do-while` loops
  input and output redirections, 169–170
  overview of, 160–161
  `RepeatAdditionQuiz.java` example, 162–163
  sentinel-controlled, 168–170
  syntax of, 160
Whitespace
  characters, 134
  as delimiter in token reading methods, 484
Wildcard import, 38
Wildcards, for specifying range of generic types, 761–764
Windows. *see* Frames (windows)
Windows OSs, 9
Wireless networking, 6
Worst-case input
  heap sorts and, 906
  measuring algorithm efficiency, 840, 854
  quick sort and, 899
Wrapper classes
  automatic conversion between primitive types and wrapper class types, 753
  `File` class as, 47
  numeric, 526
  primitive types and, 382–386
Wrapping lines of text or words, 656, 658
Write-only streams, 711. *see also* `OutputStream` classes

X

`Xlint:unchecked` error, compile time errors, 760

CHAPTER

31

ADVANCED JAVAFX AND FXML

Objectives

- To specify styles for UI nodes using JavaFX CSS (§31.2).
- To create quadratic curve, cubic curve, and path using the `QuadCurve`, `CubicCurve`, and `Path` classes (§31.3).
- To translate, rotate, and scale to perform coordinate transformations for nodes (§31.4).
- To define a shape's border using various types of strokes (§31.5).
- To create menus using the `Menu`, `MenuItem`, `CheckMenuItem`, and `RadioMemuItem` classes (§31.6).
- To create context menus using the `ContextMenu` class (§31.7).
- To use `SplitPane` to create adjustable horizontal and vertical panes (§31.8).
- To create tab panes using the `TabPane` control (§31.9).
- To create and display tables using the `TableView` and `TableColumn` classes (§31.10).
- To create JavaFX user interfaces using FMXL and the visual Scene Builder (§31.11).

31.1 Introduction

Key Point

JavaFX can be used to develop comprehensive rich Internet applications.

Chapters 14–16 introduced basics of JavaFX, event-driven programming, animations, and simple UI controls. This chapter introduces some advanced features for developing comprehensive GUI applications.

31.2 JavaFX CSS

Key Point

JavaFX cascading style sheets can be used to specify styles for UI nodes.

JavaFX cascading style sheets are based on CSS with some extensions. CSS defines the style for webpages. It separates the contents of webpages from its style. JavaFX CSS can be used to define the style for the UI and separates the contents of the UI from the style. You can define the look and feel of the UI in a JavaFX CSS file and use the style sheet to set the color, font, margin, and border of the UI components. A JavaFX CSS file makes it easy to modify the style without modifying the Java source code.

A JavaFX style property is defined with a prefix `-fx-` to distinquish it from a property in CSS. All the available JavaFX properties are defined in http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html. Listing 31.1 gives an example of a style sheet.

LISTING 31.1 mystyle.css

```
.plaincircle {
  -fx-fill: white;
  -fx-stroke: black;
}
.circleborder {
  -fx-stroke-width: 5;
  -fx-stroke-dash-array: 12 2 4 2;
}
.border {
  -fx-border-color: black;
  -fx-border-width: 5;
}
#redcircle {
  -fx-fill: red;
  -fx-stroke: red;
}
#greencircle {
  -fx-fill: green;
  -fx-stroke: green;
}
```

A style sheet uses the style class or style id to define styles. Multiple style classes can be applied to a single node, and a style id to a unique node. The syntax `.styleclass` defines a style class. Here, the style classes are named `plaincircle`, `circleborder`, and `circleborder`. The syntax `#styleid` defines a style id. Here, the style ids are named `redcircle` and `greencircle`.

Each node in JavaFX has a `styleClass` variable of the `List<String>` type, which can be obtained from invoking `getStyleClass()`. You can add multiple style classes to a node and only one id to a node. Each node in JavaFX has an `id` variable of the `String` type, which can be set using the `setID(String id)` method. You can set only one id to a node.

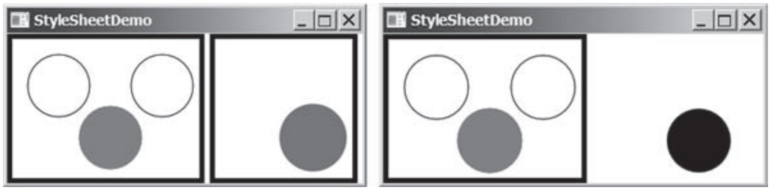
The `Scene` and `Parent` classes have the `stylesheets` property, which can be obtained from invoking the `getStylesheets()` method. This property is of the

`ObservableList<String>` type. You can add multiple style sheets into this property. You can load a style sheet into a `Scene` or a `Parent`. Note that `Parent` is the superclass for containers and UI control.

Listing 31.2 gives an example that uses the style sheet defined in Listing 31.1.

LISTING 31.2 `StyleSheetDemo.java`

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class StyleSheetDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    HBox hBox = new HBox(5);
    Scene scene = new Scene(hBox, 300, 250);
    scene.getStylesheets().add("mystyle.css"); // Load the stylesheet

    Pane pane1 = new Pane();
    Circle circle1 = new Circle(50, 50, 30);
    Circle circle2 = new Circle(150, 50, 30);
    Circle circle3 = new Circle(100, 100, 30);
    pane1.getChildren().addAll(circle1, circle2, circle3);
    pane1.getStyleClass().add("border");

    circle1.getStyleClass().add("plaincircle"); // Add a style class
    circle2.getStyleClass().add("plaincircle"); // Add a style class
    circle3.setId("redcircle"); // Add a style id

    Pane pane2 = new Pane();
    Circle circle4 = new Circle(100, 100, 30);
    circle4.getStyleClass().addAll("circleborder", "plainCircle");
    circle4.setId("greencircle"); // Add a style class
    pane2.getChildren().add(circle4);
    pane2.getStyleClass().add("border");

    hBox.getChildren().addAll(pane1, pane2);

    primaryStage.setTitle("StyleSheetDemo"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window
  }
}
```



(a) (b)

FIGURE 31.1 The style sheet is used to style the nodes in the scene.

The program loads the style sheet from the file `mystyle.css` by adding it to the `stylesheets` property (line 13). The file should be placed in the same directory with the source code for it to run correctly. After the style sheet is loaded, the program sets the style class `plaincircle` for `circle1` and `circle2` (lines 22 and 23) and sets the style id `redcircle` for `circle3` (line 24). The program sets style classes `circleborder` and `plaincircle` and an id `greencircle` for `circle4` (lines 28 and 29). The style class `border` is set for both `pane1` and `pane2` (lines 20 and 31).

The style sheet is set in the scene (line 13). All the nodes inside the scene can use this style sheet. What would happen if line 13 is deleted and the following line is inserted after line 15?

```
pane1.getStylesheets().add("mystyle.css");
```

In this case, only `pane1` and the nodes inside `pane1` can access the style sheet, but `pane2` and `circle4` cannot use this style sheet. Therefore, everything in `pane1` is displayed the same as before the change, and `pane2` and `circle4` are displayed without applying the style class and id, as shown in Figure 31.1b.

Note the style class `plaincircle` and id `greencircle` both are applied to `circle4` (lines 28 and 29). `plaincircle` sets `fill` to white and `greencircle` sets `fill` to green. The property settings in id take precedence over the ones in classes. Thus, `circle4` is displayed in green in this program.

Check Point

31.2.1 How do you load a style sheet to a `Scene` or a `Parent`? Can you load multiple style sheets?

31.2.2 If a style sheet is loaded from a node, can the pane and all its containing nodes access the style sheet?

31.2.3 Can a node add multiple style classes? Can a node set multiple style ids?

31.2.4 If the same property is defined in both a style class and a style id and applied to a node, which one has the precedence?

31.3 `QuadCurve`, `CubicCurve`, and `Path`

Key Point

JavaFX provides the **`QuadCurve`**, **`CubicCurve`**, *and* **`Path`** *classes for creating advanced shapes.*

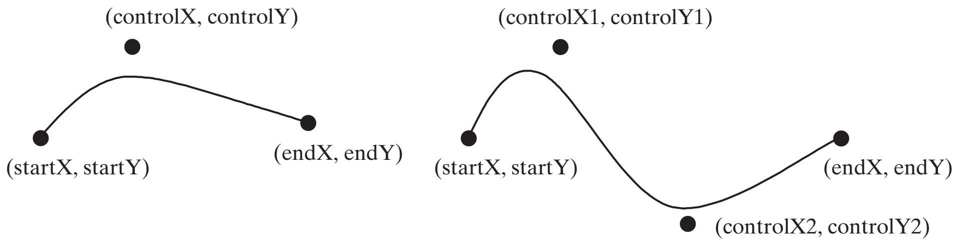
Section 14.11 introduced drawing simple shapes using the `Line`, `Rectangle`, `Circle`, `Ellipse`, `Arc`, `Polygon`, and `Polyline` classes. This section introduces drawing advanced shapes using the `CubicCurve`, `QuadCurve`, and `Path` classes.

31.3.1 `QuadCurve` and `CubicCurve`

JavaFX provides the `QuadCurve` and `CubicCurve` classes for modeling quadratic curves and cubic curves. A quadratic curve is mathematically defined as a quadratic polynomial. To create a `QuadCurve`, use its no-arg constructor or the following constructor:

```
QuadCurve(double startX, double startY,
  double controlX, double controlY, double endX, double endY)
```

where (`startX`, `startY`) and (`endX`, `endY`) specify two endpoints and (`controlX`, `controlY`) is a control point. The control point is usually not on the curve instead of defining the trend of the curve, as shown in Figure 31.2a. Figure 31.3 shows the UML diagram for the `QuadCurve` class.



(a) (b)

Figure 31.2 (a) A quadratic curve is specified using three points. (b) A cubic curve is specified using four points.

`javafx.scene.shape.QuadCurve`	The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
`-startX: DoubleProperty`	The x-coordinate of the start point (default 0).
`-startY: DoubleProperty`	The y-coordinate of the start point (default 0).
`-endX: DoubleProperty`	The x-coordinate of the end point (default 0).
`-endY: DoubleProperty`	The y-coordinate of the end point (default 0).
`-controlX: DoubleProperty`	The x-coordinate of the control point (default 0).
`-controlY: DoubleProperty`	The y-coordinate of the control point (default 0).
`+QuadCurve()`	Creates an empty quad curve.
`+QuadCurve(startX: double, startY: double, controlX: double, controlY: double, endX: double, endY: double)`	Creates a quad curve with the specified arguments.

Figure 31.3 `QuadCurve` defines a quadratic curve.

A cubic curve is mathematically defined as a cubic polynomial. To create a `CubicCurve`, use its no-arg constructor or the following constructor:

```
CubicCurve(double startX, double startY, double controlX1,
  double controlY1, double controlX2, double controlY2,
  double endX, double endY)
```

where (`startX`, `startY`) and (`endX`, `endY`) specify two endpoints and (`controlX1`, `controlY1`) and (`controlX2`, `controlY2`) are two control points. The control points are usually not on the curve, instead define the trend of the curve, as shown in Figure 31.2b. Figure 31.4 shows the UML diagram for the `CubicCurve` class.

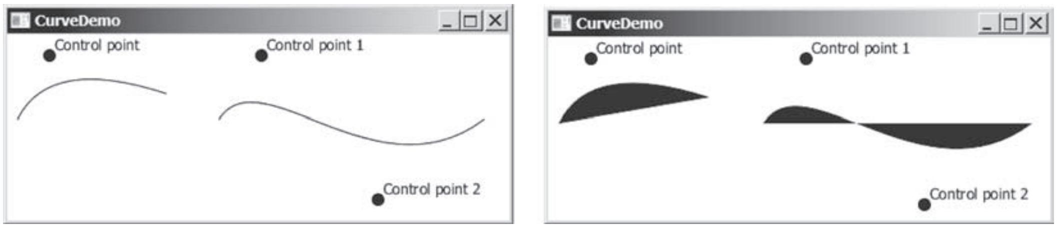
`javafx.scene.shape.CubicCurve`	The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
`-startX: DoubleProperty`	The x-coordinate of the start point (default 0).
`-startY: DoubleProperty`	The y-coordinate of the start point (default 0).
`-endX: DoubleProperty`	The x-coordinate of the end point (default 0).
`-endY: DoubleProperty`	The y-coordinate of the end point (default 0).
`-controlX1: DoubleProperty`	The x-coordinate of the first control point (default 0).
`-controlY1: DoubleProperty`	The y-coordinate of the first control point (default 0).
`-controlX2: DoubleProperty`	The x-coordinate of the second control point (default 0).
`-controlY2: DoubleProperty`	The y-coordinate of the second control point (default 0).
`+CubicCurve()`	Creates an empty cubic curve.
`+CubicCurve(startX: double, startY: double, controlX1: double, controlY1: double, controlX2: double, controlY2: double, endX: double, endY: double)`	Creates a cubic curve with the specified arguments.

Figure 31.4 `CubicCurve` defines a quadratic curve.

Listing 31.3 gives a program that demonstrates how to draw quadratic and cubic curves. Figure 31.5a shows a sample run of the program.

LISTING 31.3 CurveDemo.java

```
 1  import javafx.application.Application;
 2  import javafx.scene.Scene;
 3  import javafx.scene.layout.Pane;
 4  import javafx.scene.text.Text;
 5  import javafx.scene.shape.Circle;
 6  import javafx.scene.paint.Color;
 7  import javafx.scene.shape.*;
 8  import javafx.stage.Stage;
 9
10  public class CurveDemo extends Application {
11    @Override // Override the start method in the Application class
12    public void start(Stage primaryStage) {
13      Pane pane = new Pane();
14
15      // Create a QuadCurve
16      QuadCurve quadCurve = new QuadCurve(10, 80, 40, 20, 150, 56);
17      quadCurve.setFill(Color.WHITE);
18      quadCurve.setStroke(Color.BLACK);
19
20      pane.getChildren().addAll(quadCurve, new Circle(40, 20, 6),
21        new Text(40 + 5, 20 - 5, "Control point"));
22
23      // Create a CubicCurve
24      CubicCurve cubicCurve = new CubicCurve
25        (200, 80, 240, 20, 350, 156, 450, 80);
26      cubicCurve.setFill(Color.WHITE);
27      cubicCurve.setStroke(Color.BLACK);
28
29      pane.getChildren().addAll(cubicCurve, new Circle(240, 20, 6),
30        new Text(240 + 5, 20 – 5, "Control point 1"),
31        new Circle(350, 156, 6),
32        new Text(350 + 5, 156 – 5, "Control point 2"));
33
34      Scene scene = new Scene(pane, 300, 250);
35      primaryStage.setTitle("CurveDemo"); // Set the window title
36      primaryStage.setScene(scene); // Place the scene in the window
37      primaryStage.show(); // Display the window
38    }
39  }
```



(a) (b)

FIGURE 31.5 You can draw quadratic and cubic curves using `QuadCurve` and `CubicCurve`.

The program creates a `QuadCurve` with the specified start, control, and end points (line 16) and places the `QuadCurve` to the pane (line 20). To illustrate the control point, the program also displays the control point as a solid circle (line 21).

The program creates a **CubicCurve** with the specified start, first control, second control, and end points (lines 24 and 25) and places the **CubicCurve** to the pane (line 29). To illustrate the control points, the program also displays the control points in the pane (lines 29–32).

Note the curves are filled with color. The program sets the color to white and stroke to black in order to display the curves (lines 17 and 18, 26 and 27). If these code lines are removed from the program, the sample run would look like the one in Figure 31.5b.

31.3.2 Path

The **Path** class models an arbitrary geometric path. A path is constructed by adding path elements into the path. The **PathElement** is the root class for the path elements **MoveTo**, **HLineTo**, **VLineTo**, **LineTo**, **ArcTo**, **QuadCurveTo**, **CubicCurveTo**, and **ClosePath**.

You can create a **Path** using its no-arg constructor. The process of the path construction can be viewed as drawing with a pen. The path does not have a default initial position. You need to set an initial position by adding a **MoveTo(startX, startY)** path element to the path. Adding a **HLineTo(newX)** element draws a horizontal line from the current position to the new *x*-coordinate. Adding a **VLineTo(newY)** element draws a vertical line from the current position to the new *y*-coordinate. Adding a **LineTo(newX, newY)** element draws a line from the current position to the new position. Adding an **ArcTo(radiusX, radiusY, xAxisRotation, newX, newY, largeArcFlag, sweepArcFlag)** element draws an arc from the previous position to the new position with the specified radius. Adding a **QuadCurveTo(controlX, controlY, newX, newY)** element draws a quadratic curve from the previous position to the new position with the specified control point. Adding a **CubicCurveTo(controlX1, controlY1, controlX2, controlY2, newX, newY)** element draws a cubic curve from the previous position to the new position with the specified control points. Adding a **ClosePath()** element closes the path by drawing a line that connects the starting point with the end point of the path.

Listing 31.4 gives an example that creates a path. A sample run of the program is shown in Figure 31.6.

LISTING 31.4 PathDemo.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.*;
import javafx.stage.Stage;

public class PathDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    Pane pane = new Pane();

    // Create a Path
    Path path = new Path();
    path.getElements().add(new MoveTo(50.0, 50.0));
    path.getElements().add(new HLineTo(150.5));
    path.getElements().add(new VLineTo(100.5));
    path.getElements().add(new LineTo(200.5, 150.5));

    ArcTo arcTo = new ArcTo(45, 45, 250, 100.5,
      false, true);
    path.getElements().add(arcTo);

    path.getElements().add(new QuadCurveTo(50, 50, 350, 100));
    path.getElements().add(
      new CubicCurveTo(250, 100, 350, 250, 450, 10));
27
28      path.getElements().add(new ClosePath());
29
30      pane.getChildren().add(path);
31      path.setFill(null);
32      Scene scene = new Scene(pane, 300, 250);
33      primaryStage.setTitle("PathDemo"); // Set the window title
34      primaryStage.setScene(scene); // Place the scene in the window
35      primaryStage.show(); // Display the window
36    }
37  }
```

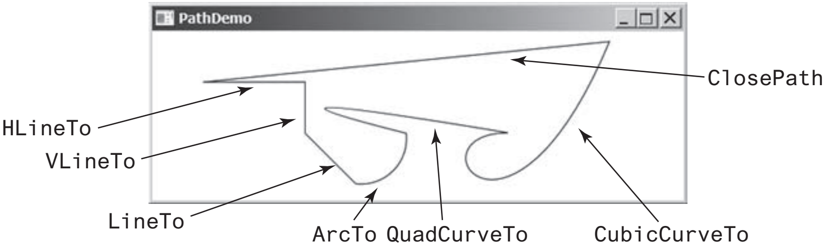


FIGURE 31.6 You can draw a path by adding path elements.

The program creates a `Path` (line 14), moves its position (line 15), and adds a horizontal line (line 16), a vertical line (line 17), and a line (line 18). The `getElements()` method returns an `ObservableList<PathElement>`.

The program creates an `ArcTo` object (lines 20 and 21). The `ArcTo` class contains the `largeArcFlag` and `sweepFlag` properties. By default, these property values are `false`. You may set these properties to `ture` to display a large arc in the opposite direction.

The program adds a quadratic curve (line 24) and a cubic curve (lines 25 and 26) and closes the path (line 28).

By default, the path is not filled. You may change the `fill` property in the path to specify a color to fill the path.

Check Point

31.3.1 Create a `QuadCurve` with starting point (100, 75.5), control point (40, 55.5), and end point (56, 80). Set its `fill` property to white and `stroke` to green.

31.3.2 Create `CubicCurve` object with starting point (100, 75.5), control point 1 (40, 55.5), control point 2 (78.5, 25.5), and end point (56, 80). Set its `fill` property to white and `stroke` to green.

31.3.3 Does a path have a default initial position? How do you set a position for a path?

31.3.4 How do you close a path?

31.3.5 How do you display a filled path?

31.4 Coordinate Transformations

Key Point

JavaFX supports coordinate transformations using translation, rotation, and scaling.

You have used the `rotate` method to rotate a node. You can also perform translations and scaling.

31.4.1 Translations

You can use the `setTranslateX(double x)`, `setTranslateY(double y)`, and `setTranslateZ(double z)` methods in the `Node` class to translate the coordinates for a node. For example, `setTranslateX(5)` moves the node 5 pixels to the right and `setTranslateY(-10)` 10 pixels up from the previous position. Figure 31.7 shows a rectangle displayed before and after applying translation. After invoking `rectangle.setTranslateX(-6)` and `rectangle.setTranslateY(4)`, the rectangle is moved 6 pixels to the left and 4 pixels down from the previous position. Note the coordinate transformation using translation, rotation, and scaling does not change the contents of the shape being transferred. For example, if a rectangle's *x* is 30 and width is 100, after applying transformations to the rectangle, its *x* is still 30 and width is still 100.

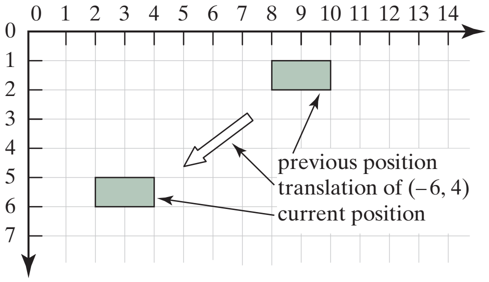


FIGURE 31.7 After applying translation of (−6, 4), the rectangle is moved by the specified distance relative to the previous position.

LISTING 31.5 `TranslationDemo.java`

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class TranslationDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    Pane pane = new Pane();

    double x = 10;
    double y = 10;
    java.util.Random random = new java.util.Random();
    for (int i = 0; i < 10; i++) {
      Rectangle rectangle = new Rectangle(10, 10, 50, 60);
      rectangle.setFill(Color.WHITE);
      rectangle.setStroke(Color.color(random.nextDouble(),
        random.nextDouble(), random.nextDouble()));
      rectangle.setTranslateX(x += 20);
      rectangle.setTranslateY(y += 5);
      pane.getChildren().add(rectangle);
    }

    Scene scene = new Scene(pane, 300, 250);
    primaryStage.setTitle("TranslationDemo"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window
  }
}
```

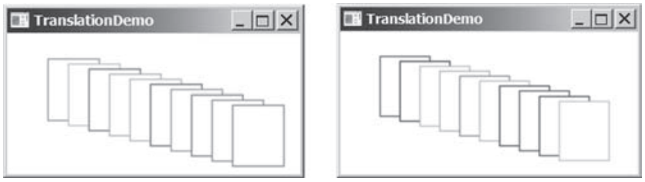


FIGURE 31.8 The rectangles are displayed successively in new locations.

The program repeatedly creates 10 rectangles (line 17). For each rectangle, it sets its `fill` property to white (line 18) and its `stroke` property to a random color (lines 19 and 20), and translates it to a new location (lines 21 and 22). The variables `x` and `y` are used to set the `translateX` and `translateY` properties. These two variable values are changed every time it is applied to a rectangle (see Figure 31.8).

31.4.2 Rotations

Rotation was introduced in Chapter 14. This section discusses it in more depth. You can use the `rotate(double theta)` method in the `Node` class to rotate a node by `theta` degrees from its pivot point clockwise, where `theta` is a double value in degrees. The pivot point is automatically computed based on the bounds of the node. For a circle, ellipse, and a rectangle, the pivot point is the center point of these nodes. For example, `rectangle.rotate(45)` rotates the rectangle 45 degrees clockwise along the eastern direction from the center, as shown in Figure 31.9.

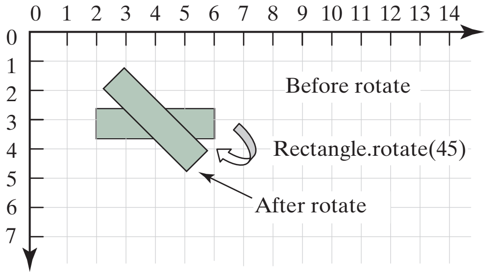


FIGURE 31.9 After performing `rectangle.rotate(45)`, the rectangle is rotated in 45 degrees from the center.

Listing 31.6 gives a program that demonstrates the effect of rotation of coordinates. Figure 31.10 shows a sample run of the program.

LISTING 31.6 `RotateDemo.java`

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class RotateDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    Pane pane = new Pane();
    java.util.Random random = new java.util.Random();
13     // The radius of the circle for anchoring rectangles
14     double radius = 90;
15     double width = 20; // Width of the rectangle
16     double height = 40; // Height of the rectangle
17     for (int i = 0; i < 8; i++) {
18       // Center of a rectangle
19       double x = 150 + radius * Math.cos(i * 2 * Math.PI / 8);
20       double y = 150 + radius * Math.sin(i * 2 * Math.PI / 8);
21       Rectangle rectangle = new Rectangle(
22         x - width / 2, y - height / 2, width, height);
23       rectangle.setFill(Color.WHITE);
24       rectangle.setStroke(Color.color(random.nextDouble(),
25         random.nextDouble(), random.nextDouble()));
26       rectangle.setRotate(i * 360 / 8); // Rotate the rectangle
27       pane.getChildren().add(rectangle);
28     }
29
30     Scene scene = new Scene(pane, 300, 300);
31     primaryStage.setTitle("RotateDemo"); // Set the window title
32     primaryStage.setScene(scene); // Place the scene in the window
33     primaryStage.show(); // Display the window
34   }
35 }
```

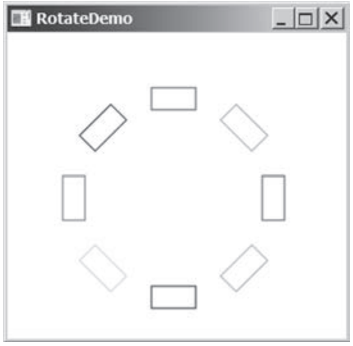


FIGURE 31.10 The `rotate` method rotates a node.

The program creates eight rectangles in a loop (lines 17–28). The center of each rectangle is located on the circle centered as (150, 150) (lines 19 and 20). A rectangle is created by specifying its upper left corner position with width and height (lines 21 and 22). The rectangle is rotated in line 26 and added to the pane in line 27.

31.4.3 Scaling

You can use the `setScaleX(double sx)`, `setScaleY(double sy)`, and `setScaleY(double sy)` methods in the `Node` class to specify a scaling factor. The node will appear larger or smaller depending on the scaling factor. Scaling alters the coordinate space of the node such that each unit of distance along the axis is multiplied by the scale factor. As with rotation transformations, scaling transformations are applied to enlarge or shrink the node around the pivot point. For a node of the rectangle shape, the pivot point is the center of the rectangle. For example, if you apply a scaling factor ($x = 2$, $y = 2$), the entire rectangle including the stroke will double in size, growing to the left, right, up, and down from the center, as shown in Figure 31.11.

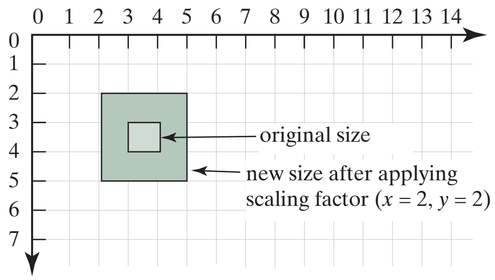


FIGURE 31.11 After applying scaling ($x = 2$, $y = 2$), the node is doubled in size.

Listing 31.7 gives a program that demonstrates the effect of using scaling. Figure 31.12 shows a sample run of the program.

LISTING 31.7 ScaleDemo.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;
import javafx.scene.shape.Polyline;
import javafx.stage.Stage;

public class ScaleDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a polyline to draw a sine curve
    Polyline polyline = new Polyline();
    for (double angle = -360; angle <= 360; angle++) {
      polyline.getPoints().addAll(
        angle, Math.sin(Math.toRadians(angle)));
    }
    polyline.setTranslateY(100);
    polyline.setTranslateX(200);
    polyline.setScaleX(0.5);
    polyline.setScaleY(50);
    polyline.setStrokeWidth(1.0 / 25);

    // Draw x-axis
    Line line1 = new Line(10, 100, 420, 100);
    Line line2 = new Line(420, 100, 400, 90);
    Line line3 = new Line(420, 100, 400, 110);

    // Draw y-axis
    Line line4 = new Line(200, 10, 200, 200);
    Line line5 = new Line(200, 10, 190, 30);
    Line line6 = new Line(200, 10, 210, 30);

    // Draw x, y axis labels
    Text text1 = new Text(380, 70, "X");
    Text text2 = new Text(220, 20, "Y");

    // Add nodes to a pane
    Pane pane = new Pane();
    pane.getChildren().addAll(polyline, line1, line2, line3, line4,
      line5, line6, text1, text2);

43       Scene scene = new Scene(pane, 450, 200);
44       primaryStage.setTitle("ScaleDemo"); // Set the window title
45       primaryStage.setScene(scene); // Place the scene in the window
46       primaryStage.show(); // Display the window
47     }
48   }
```

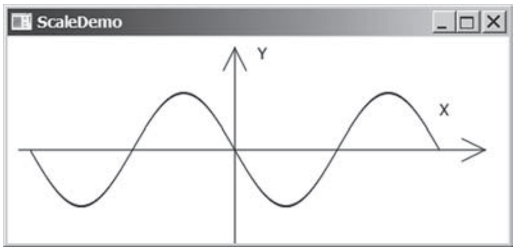


FIGURE 31.12 The `scale` method scales the coordinates in the node.

The program creates a polyline (line 13) and adds the points for a sine curve into the polyline (lines 14–17). Since $|\sin(x)| <= 1$, the y-coordinates are too small. To see the sine curve, the program scales the y-coordinates up by 50 times (line 21) and shrinks the x-coordinates by half (line 20).

Note scaling also causes the stroke width to change. To compensate it, the stroke width is purposely set to 1.0 / 25 (line 22).

Check Point

31.4.1 Can you perform a coordinate transformation on any node? Does a coordinate transformation change the contents of a `Shape` object?

31.4.2 Does the method `setTranslateX(6)` move the node's x-coordinate to 6? Does the method `setTranslateX(6)` move the node's x-coordinate 6 pixel right from its current location?

31.4.3 Does the method `rotate(Math.PI / 2)` rotate a node 90 degrees? Does the method `rotate(90)` rotate a node 90 degrees?

31.4.4 How is the pivot point determined for performing a rotation?

31.4.5 What method do you use to scale a node two times on its x-axis?

31.5 Strokes

Stroke defines a shape's border line style.

Key Point

JavaFX allows you to specify the attributes of a shape's boundary using the methods in Figure 31.13.

`javafx.scene.shape.Shape`	
`+setStroke(paint: Paint): void`	Sets a paint for the stroke.
`+setStrokeWidth(width: double): void`	Sets a width for the stroke (default 1).
`+setStrokeType(type: StrokeType): void`	Sets a type for the stroke to indicate whether the stroke is placed inside, centered, or outside of the border (default: CENTERED).
`+setStrokeLineCap(type: StrokeLineCap): void`	Specifies the end cap style for the stroke (default: BUTT).
`+setStrokeLineJoin(type: StrokeLineJoin): void`	Specifies how two line segments are joined (default: MITER).
`+getStrokeDashArray(): ObservableList<Double>`	Returns a list that specifies a dashed pattern for line segments.
`+setStrokeDashOffset(distance: double): void`	Specifies the offset to the first segment in the dashed pattern.

FIGURE 31.13 The `Shape` class contains the methods for setting stroke properties.

The `setStroke(paint)` method sets a paint for the stroke. The width of the stroke can be specified using the `setStrokeWidth(width)` method.

The `setStrokeType(type)` method sets a type for the stroke. The type defines whether the stroke is inside, outside, or in the center of the border using the constants `StrokeType.INSIDE`, `StrokeType.OUTSIDE`, or `StrokeType.CENTERED` (default), as shown in Figure 31.14.

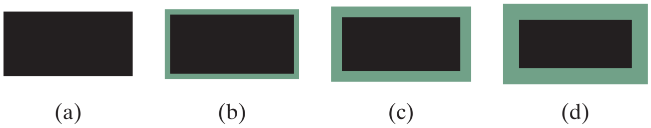

(a) (b) (c) (d)

FIGURE 31.14 (a) No stroke is used. (b) A stroke is placed inside the border. (c) A stroke is placed in the center of the border. (d) A stroke is placed outside of the border.

Note for the centered style, the stroke is applied by extending the boundary of the node by a distance of half of the `strokeWidth` on either side (inside and outside) of the boundary.

The `setStrokeLineCap(capType)` method sets an end cap style for the stroke. The styles are defined as `StrokeLineCap.BUTT` (default), `StrokeLineCap.ROUND`, and `StrokeLineCap.SQUARE`, as illustrated in Figure 31.15. The `BUTT` stroke ends an unclosed path with no added decoration. The `ROUND` stroke ends an unclosed side of a path with an added half circle whose radius is half of the stroke width. The `SQUARE` stroke ends an unclosed side of a path with an added square that extends half of the stroke width.

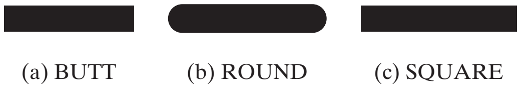

(a) BUTT (b) ROUND (c) SQUARE

FIGURE 31.15 (a) No decoration for a BUTT line cap. (b) A half circle is added to an unclosed path. (c) A square with half of the stroke width is extended to an unclosed path.

The `setStrokeLineJoin` method defines the decoration applied where path segments meet. You can specify three types of line join using the constants `StrokeLineJoin.MITER` (default), `StrokeLineJoin.BEVEL`, and `StrokeLineJoin.ROUND`, as shown in Figure 31.16.

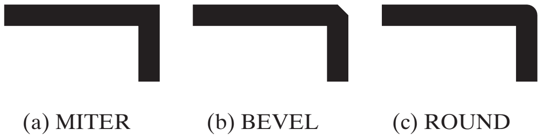

(a) MITER (b) BEVEL (c) ROUND

FIGURE 31.16 Path segments can be joined in three ways: (a) MITER, (b) BEVEL, and (c) ROUND.

The `Shape` class has a property named `strokeDashArray` of the `ObservableList<Double>` type. This property is used to define a dashed pattern for the stroke. Alternate numbers in the list specify the lengths of the opaque and transparent segments of the dashes. For example, the list `[10.0, 20.0, 30.0, 40.0]` specifies a pattern as shown in Figure 31.17.

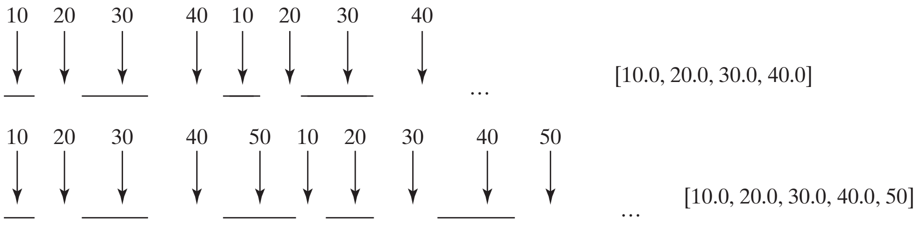


FIGURE 31.17 The numbers in the list specify the opaque and transparent segments of the stroke alternately.

The `setStrokeDashOffset(distance)` method defines the offset to the first segment in the dash pattern. Figure 31.18 illustrates the offset 5 for the dash list `[10.0, 20.0, 30.0, 40.0]`.

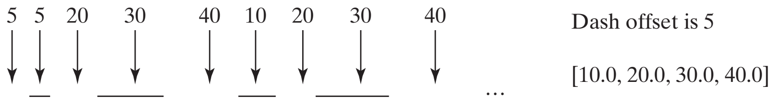


FIGURE 31.18 The dash offset specifies on offset for the first segment.

Listing 31.8 gives a program that demonstrates the methods to set attributes for a stroke. Figure 31.19 shows a sample run of the program.

LISTING 31.8 StrokeDemo.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Rectangle;
import javafx.scene.shape.*;

public class StrokeDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    Rectangle rectangle1 = new Rectangle(20, 20, 70, 120);
    rectangle1.setFill(Color.WHITE);
    rectangle1.setStrokeWidth(15);
    rectangle1.setStroke(Color.ORANGE);

    Rectangle rectangle2 = new Rectangle(20, 20, 70, 120);
    rectangle2.setFill(Color.WHITE);
    rectangle2.setStrokeWidth(15);
    rectangle2.setStroke(Color.ORANGE);
    rectangle2.setTranslateX(100);
    rectangle2.setStrokeLineJoin(StrokeLineJoin.BEVEL);

    Rectangle rectangle3 = new Rectangle(20, 20, 70, 120);
    rectangle3.setFill(Color.WHITE);
    rectangle3.setStrokeWidth(15);
    rectangle3.setStroke(Color.ORANGE);
    rectangle3.setTranslateX(200);
    rectangle3.setStrokeLineJoin(StrokeLineJoin.ROUND);

    Line line1 = new Line(320, 20, 420, 20);
    line1.setStrokeLineCap(StrokeLineCap.BUTT);
    line1.setStrokeWidth(20);

    Line line2 = new Line(320, 70, 420, 70);
    line2.setStrokeLineCap(StrokeLineCap.ROUND);
    line2.setStrokeWidth(20);

    Line line3 = new Line(320, 120, 420, 120);
    line3.setStrokeLineCap(StrokeLineCap.SQUARE);
    line3.setStrokeWidth(20);

    Line line4 = new Line(460, 20, 560, 120);
    line4.getStrokeDashArray().addAll(10.0, 20.0, 30.0, 40.0);

46      Pane pane = new Pane();
47      pane.getChildren().addAll(rectangle1, rectangle2, rectangle3,
48        line1, line2, line3, line4);
49
50      Scene scene = new Scene(pane, 610, 180);
51      primaryStage.setTitle("StrokeDemo"); // Set the window title
52      primaryStage.setScene(scene); // Place the scene in the window
53      primaryStage.show(); // Display the window
54    }
55
56    // Launch the program from command-line
57    public static void main(String[] args) {
58      launch(args);
59    }
60  }
```

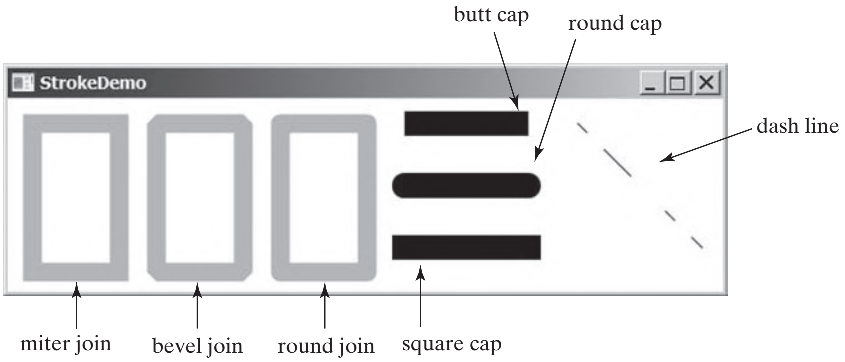


FIGURE 31.19 You can specify the attributes for strokes.

The program creates three rectangles (lines 12–29). Rectangle 1 uses default miter join, rectangle 2 uses bevel join (line 22), and rectangle 3 uses round join (line 29).

The program creates three lines with butt, round, and square end cap (lines 31–41).

The program creates a line and sets dash pattern for this line (line 44). Note the `strokeDashArray` property is of the `ObservableList<Double>` type. You have to add `Double` values to the list. Adding a number such as 10 would cause an error.

Check Point

31.5.1 Are the methods for setting a stroke and its attributes defined in the `Node` or `Shape` class?

31.5.2 How do you set a stroke width to `3` pixels?

31.5.3 What are the stroke types? What is the default stroke type? How do you set a stroke type?

31.5.4 What are the stroke line join types? What is the default stroke line join type? How do you set a stroke line join type?

31.5.5 What are the stroke cap types? What is the default stroke cap type? How do you set a stroke cap type?

31.5.6 How do you specify a dashed pattern for strokes?

31.6 Menus

Key Point

You can create menus in JavaFX.

Menus make selection easier and are widely used in window applications. JavaFX provides five classes that implement menus: `MenuBar`, `Menu`, `MenuItem`, `CheckMenuItem`, and `RadioButtonMenuItem`.

`MenuBar` is a top-level menu component used to hold the menus. A menu consists of menu items that the user can select (or toggle on or off). A menu item can be an instance of `MenuItem`, `CheckMenuItem`, or `RadioButtonMenuItem`. `Menu` items can be associated with nodes and keyboard accelerators.

31.6.1 Creating Menus

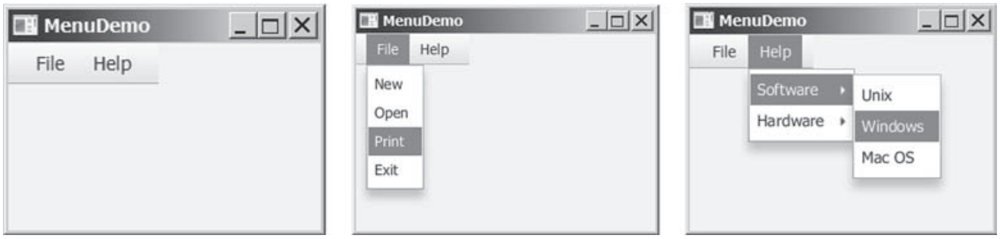
The sequence of implementing menus in JavaFX is as follows:

1. Create a menu bar and add it to a pane. For example, the following code creates a pane and a menu bar, and adds the menu bar to the pane:

```
MenuBar menuBar = new MenuBar();
Pane pane = new Pane();
pane.getChildren().add(menuBar);
```

2. Create menus and add them under the menu bar. For example, the following creates two menus and adds them to a menu bar, as shown in Figure 31.20a:

```
Menu menuFile = new Menu("File");
Menu menuHelp = new Menu("Help");
menuBar.getMenus().addAll(menuFile, menuHelp);
```



(a) (b) (c)

FIGURE 31.20 (a) The menus are placed under a menu bar. (b) Clicking a menu on the menu bar reveals the items under the menu. (c) Clicking a menu item reveals the submenu items under the menu item.

3. Create menu items and add them to the menus.

```
menuFile.getItems().addAll(new MenuItem("New"),
  new MenuItem("Open"), new MenuItem("Print"),
  new MenuItem("Exit"));
```

This code adds the menu items New, Open, Print, and Exit, in this order, to the File menu, as shown in Figure 31.20b.

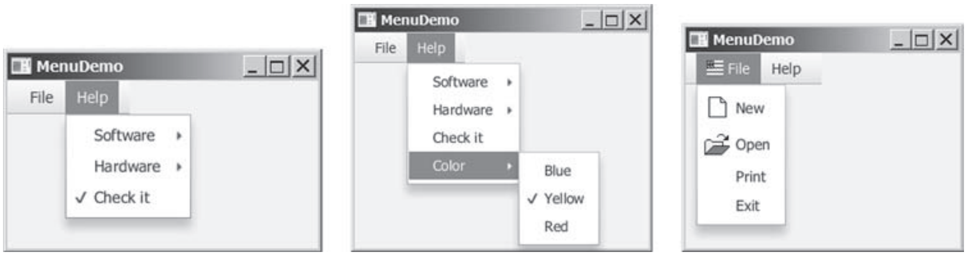
3.1. Creating submenu items.
You can also embed menus inside menus so the embedded menus become submenus. Here is an example:

```
Menu softwareHelpSubMenu = new Menu("Software");
Menu hardwareHelpSubMenu = new Menu("Hardware");
menuHelp.getItems().add(softwareHelpSubMenu);
menuHelp.getItems().add(hardwareHelpSubMenu);
softwareHelpSubMenu.getItems().add(new MenuItem("Unix"));
softwareHelpSubMenu.getItems().add(new MenuItem("Windows"));
softwareHelpSubMenu.getItems().add(new MenuItem("Mac OS"));
```

This code adds two submenus, `softwareHelpSubMenu` and `hardwareHelpSubMenu`, in `MenuHelp`. The menu items `Unix`, `NT`, and `Win95` are added to `softwareHelpSubMenu` (see Figure 31.20c).

3.2. Creating check-box menu items.
You can also add a `CheckMenuItem` to a `Menu`. `CheckMenuItem` is a subclass of `MenuItem` that adds a Boolean state to the `MenuItem` and displays a check when its state is true. You can click a menu item to turn it on or off. For example, the following statement adds the check-box menu item Check it (see Figure 31.21a).

```
menuHelp.getItems().add(new CheckMenuItem("Check it"));
```



(a) (b) (c)

FIGURE 31.21 (a) A check box menu item lets you check or uncheck a menu item just like a check box. (b) You can use `RadioMenuItem` to choose among mutually exclusive menu choices. (c) You can set image icons and keyboard accelerators in menus.

3.3. Creating radio menu items.
You can also add radio menu items to a menu, using the `RadioMenuItem` class. This is often useful when you have a group of mutually exclusive choices in the menu. For example, the following statements add a submenu named `Color` and a set of radio buttons for choosing a color (see Figure 31.21b):

```
RadioMenuItem rmiBlue, rmiYellow, rmiRed;
colorHelpSubMenu.getItems().add(rmiBlue =
  new RadioMenuItem("Blue"));
colorHelpSubMenu.getItems().add(rmiYellow =
  new RadioMenuItem("Yellow"));
colorHelpSubMenu.getItems().add(rmiRed =
  new RadioMenuItem("Red"));

ToggleGroup group = new ToggleGroup();
rmiBlue.setToggleGroup(group);
rmiYellow.setToggleGroup(group);
rmiRed.setToggleGroup(group);
```

4. The menu items generate `ActionEvent`. To handle `ActionEvent`, implement the `setOnAction` method.

5. **Image Icons and Keyboard Accelerators**

The `Menu`, `CheckMenuItem`, and `RadioMenuItem` are the subclasses of `MenuItem`. The `MenuItem` has a `graphic` property for specifying a node to be displayed in the menu item. Usually, the graphic is an image view. The classes `Menu`, `MenuItem`, `CheckMenuItem`, and `RadioMenuItem` have another constructor that you can use to specify a graphic. For example, the following code adds an image to the menu, menu item, check menu item, and radio menu item (see Figure 31.21c).

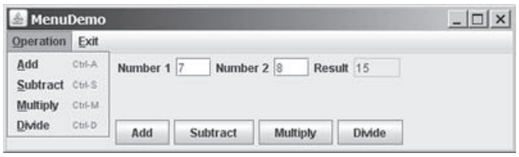
```
Menu menuFile = new Menu("File",
  new ImageView("image/usIcon.gif"));
MenuItem menuItemOpen = new MenuItem("New",
  new ImageView("image/new.gif"));
CheckMenuItem checkMenuItem = new CheckMenuItem("Check it",
  new ImageView("image/us.gif"));
RadioMenuItem rmiBlue = new RadioMenuItem("Blue",
  new ImageView("image/us.gif"));
```

6. A key accelerator lets you select a menu item directly by pressing the CTRL and the accelerator key. For example, by using the following code, you can attach the accelerator key CTRL+N to the Open menu item:

```
menuItemOpen.setAccelerator(
  KeyCombination.keyCombination("Ctrl+O"));
```

31.6.2 Example: Using Menus

This section gives an example that creates a user interface to perform arithmetic. The interface contains labels and text fields for Number 1, Number 2, and Result. The Result text field displays the result of the arithmetic operation between Number 1 and Number 2. Figure 31.22 contains a sample run of the program.



Number 1 7 Number 2 8 Result 15

Figure 31.22 Arithmetic operations can be performed by clicking buttons or by choosing menu items from the Operation menu.

Here are the major steps in the program (Listing 31.9):

1. Create a menu bar and add it into a **VBox**. Create the menus Operation and Exit, and add them to the menu bar. Add the menu items Add, Subtract, Multiply, and Divide under the Operation menu and add the menu item Close under the Exit menu.
2. Create an **HBox** to hold labels and text fields and place it into the **VBox**.
3. Create an **HBox** to hold the four buttons labeled Add, Subtract, Multiply, and Divide and place it into the **VBox**.
4. Implement the handlers to process the events from the menu items and the buttons.

Listing 31.9 MenuDemo.java

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.control.TextField;
import javafx.scene.input.KeyCombination;
```

```java
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class MenuDemo extends Application {
  private TextField tfNumber1 = new TextField();
  private TextField tfNumber2 = new TextField();
  private TextField tfResult = new TextField();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    MenuBar menuBar = new MenuBar();

    Menu menuOperation = new Menu("Operation");
    Menu menuExit = new Menu("Exit");
    menuBar.getMenus().addAll(menuOperation, menuExit);

    MenuItem menuItemAdd = new MenuItem("Add");
    MenuItem menuItemSubtract = new MenuItem("Subtract");
    MenuItem menuItemMultiply = new MenuItem("Multiply");
    MenuItem menuItemDivide = new MenuItem("Divide");
    menuOperation.getItems().addAll(menuItemAdd, menuItemSubtract,
      menuItemMultiply, menuItemDivide);

    MenuItem menuItemClose = new MenuItem("Close");
    menuExit.getItems().add(menuItemClose);

    menuItemAdd.setAccelerator(
      KeyCombination.keyCombination("Ctrl+A"));
    menuItemSubtract.setAccelerator(
      KeyCombination.keyCombination("Ctrl+S"));
    menuItemMultiply.setAccelerator(
      KeyCombination.keyCombination("Ctrl+M"));
    menuItemDivide.setAccelerator(
      KeyCombination.keyCombination("Ctrl+D"));

    HBox hBox1 = new HBox(5);
    tfNumber1.setPrefColumnCount(2);
    tfNumber2.setPrefColumnCount(2);
    tfResult.setPrefColumnCount(2);
    hBox1.getChildren().addAll(new Label("Number 1:"), tfNumber1,
      new Label("Number 2:"), tfNumber2, new Label("Result:"),
      tfResult);
    hBox1.setAlignment(Pos.CENTER);

    HBox hBox2 = new HBox(5);
    Button btAdd = new Button("Add");
    Button btSubtract = new Button("Subtract");
    Button btMultiply = new Button("Multiply");
    Button btDivide = new Button("Divide");
    hBox2.getChildren().addAll(btAdd, btSubtract, btMultiply, btDivide);
    hBox2.setAlignment(Pos.CENTER);

    VBox vBox = new VBox(10);
    vBox.getChildren().addAll(menuBar, hBox1, hBox2);
    Scene scene = new Scene(vBox, 300, 250);
    primaryStage.setTitle("MenuDemo"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window

```

```
 71       // Handle menu actions
 72       menuItemAdd.setOnAction(e -> perform('+'));
 73       menuItemSubtract.setOnAction(e -> perform('-'));
 74       menuItemMultiply.setOnAction(e -> perform('*'));
 75       menuItemDivide.setOnAction(e -> perform('/'));
 76       menuItemClose.setOnAction(e -> System.exit(0));
 77
 78       // Handle button actions
 79       btAdd.setOnAction(e -> perform('+'));
 80       btSubtract.setOnAction(e -> perform('-'));
 81       btMultiply.setOnAction(e -> perform('*'));
 82       btDivide.setOnAction(e -> perform('/'));
 83     }
 84
 85     private void perform(char operator) {
 86       double number1 = Double.parseDouble(tfNumber1.getText());
 87       double number2 = Double.parseDouble(tfNumber2.getText());
 88
 89       double result = 0;
 90       switch (operator) {
 91         case '+': result = number1 + number2; break;
 92         case '-': result = number1 - number2; break;
 93         case '*': result = number1 * number2; break;
 94         case '/': result = number1 / number2; break;
 95      }
 96
 97      tfResult.setText(result + "");
 98    }
100  }
```

The program creates a menu bar (line 22), which holds two menus: `menuOperation` and `menuExit` (lines 24–36). The `menuOperation` contains four menu items for doing arithmetic: Add, Subtract, Multiply, and Divide. The `menuExit` contains the menu item Close for exiting the program. The menu items in the Operation menu are created with keyboard accelerators (lines 38–45).

The labels and text fields are placed in an `HBox` (lines 47–54) and four buttons are placed in another `HBox` (lines 56–62). The menu bar and these two `HBoxes` are added to a `VBox` (line 65), which is placed in the scene (line 66).

The user enters two numbers in the number fields. When an operation is chosen from the menu, its result, involving two numbers, is displayed in the Result field. The user can also click the buttons to perform the same operation.

The program sets actions for the menu items and buttons in lines 72–82. The private method `perform(char operator)` (lines 85–98) retrieves operands from the text fields in Number 1 and Number 2, applies the binary operator on the operands, and sets the result in the Result text field.

Check Point

31.6.1 How do you create a menu bar, menu, menu item, check menu item, and radio menu item?

31.6.2 How do you place a menu into a menu bar? How do you place a menu item, check menu item, and radio menu item into a menu?

31.6.3 Can you place a menu item into another menu item or a check menu or a radio menu item into a menu item?

31.6.4 How do you associate an image with a menu, menu item, check menu item, and radio menu item?

31.6.5 How do you associate an accelerator CTRL+O with a menu item, check menu item, and radio menu item?

31.7 Context Menus

Key Point

You can create context menus in JavaFX.

A *context menu*, also known as a *popup menu*, is like a regular menu, but does not have a menu bar and can float anywhere on the screen. Creating a context menu is similar to creating a regular menu. First, you create an instance of `ContextMenu`, and then you can add `MenuItem`, `CheckMenuItem`, and `RadioMenuItem` to the context menu. For example, the following code creates a `ContextMenu`, then adds `MenuItems` into it:

```
ContextMenu contextMenu = new ContextMenu();
ContextMenu.getItems().add(new MenuItem("New"));
ContextMenu.getItems().add(new MenuItem("Open"));
```

A regular menu is always added to a menu bar, but a context menu is associated with a parent node and is displayed using the show method in the `ContextMenu` class. You specify the parent node and the location of the context menu, using the coordinate system of the parent like this:

```
contextMenu.show(node, x, y);
```

Customarily, you display a context menu by pointing to a GUI component and clicking a certain mouse button, the so-called popup trigger. Popup triggers are system dependent. In Windows, the context menu is displayed when the right mouse button is released. In Motif, the context menu is displayed when the third mouse button is pressed and held down.

Listing 31.10 gives an example that creates a pane. When the mouse points to the pane, clicking a mouse button displays a context menu, as shown in Figure 31.23.

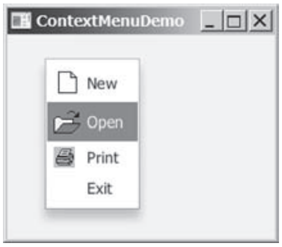


FIGURE 31.23 A context menu is displayed when the popup trigger is issued on the pane.

Here are the major steps in the program (Listing 31.10):

1. Create a context menu using `ContextMenu`. Create menu items for New, Open, Print, and Exit using `MenuItem`.
2. Add the menu items into the context menu.
3. Create a pane and place it in the scene.
4. Implement the handler to process the events from the menu items.
5. Implement the `mousePressed` handler to display the context menu.

LISTING 31.10 ContextMenuDemo.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ContextMenu;
import javafx.scene.control.MenuItem;
import javafx.scene.image.ImageView;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

```

```
 9 public class ContextMenuDemo extends Application {
10   @Override // Override the start method in the Application class
11   public void start(Stage primaryStage) {
12     ContextMenu contextMenu = new ContextMenu();
13     MenuItem menuItemNew = new MenuItem("New",
14       new ImageView("image/new.gif"));
15     MenuItem menuItemOpen = new MenuItem("Open",
16       new ImageView("image/open.gif"));
17     MenuItem menuItemPrint = new MenuItem("Print",
18       new ImageView("image/print.gif"));
19     MenuItem menuItemExit = new MenuItem("Exit");
20     contextMenu.getItems().addAll(menuItemNew, menuItemOpen,
21       menuItemPrint, menuItemExit);
22
23     Pane pane = new Pane();
24     Scene scene = new Scene(pane, 300, 250);
25     primaryStage.setTitle("ContextMenuDemo"); // Set the window title
26     primaryStage.setScene(scene); // Place the scene in the window
27     primaryStage.show(); // Display the window
28
29     pane.setOnMousePressed(
30       e -> contextMenu.show(pane, e.getScreenX(), e.getScreenY()));
31
32     menuItemNew.setOnAction(e -> System.out.println("New"));
33     menuItemOpen.setOnAction(e -> System.out.println("Open"));
34     menuItemPrint.setOnAction(e -> System.out.println("Print"));
35     menuItemExit.setOnAction(e -> System.exit(0));
36   }
37 }
```

The process of creating context menus is similar to the process for creating regular menus. To create a context menu, create a `ContextMenu` as the basis (line 12) and add `MenuItems` to it (lines 13–21).

To show a context menu, use the show method by specifying the parent node and the location for the context menu (lines 29 and 30). The show method is invoked when the context menu is triggered by a mouse click on the pane (line 30).

Check Point

31.7.1 How do you create a context menu? How do you add menu items, check menu items, and radio menu items into a context menu?

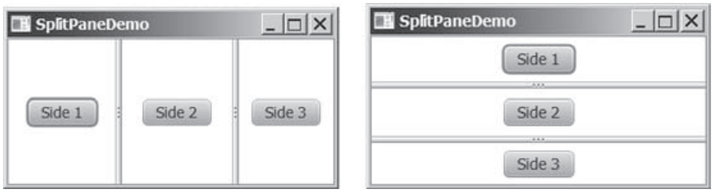
31.7.2 How do you show a context menu?

31.8 SplitPane

Key Point

The `SplitPane` *class can be used to display multiple panes and allow the user to adjust the size of the panes.*

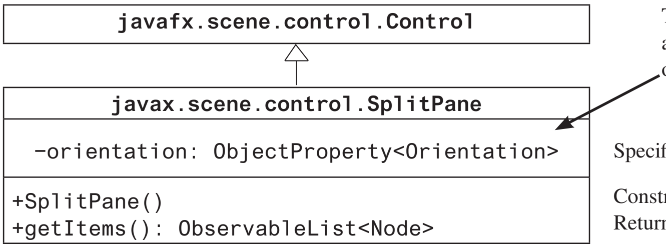
The `SplitPane` is a control that contains two components with a separate bar known as a divider, as shown in Figure 31.24.



(a) Horizontal orientation (b) Vertical orientation

Figure 31.24 `SplitPane` divides a container into two parts.

The two sides separated by the divider can appear in horizontal or vertical orientation. The divider separating two sides can be dragged to change the amount of space occupied by each side. Figure 31.25 shows the frequently used properties, constructors, and methods in `SplitPane`.


javafx.scene.control.Control
javax.scene.control.SplitPane
-orientation: ObjectProperty<Orientation>
+SplitPane()
+getItems(): ObservableList<Node>

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Specifies the orientation of the pane.

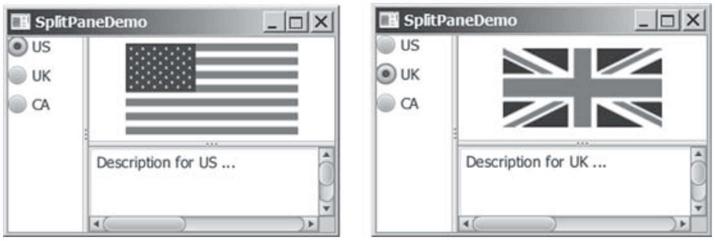
Constructs a default split pane with horizontal orientation.
Returns a list of items in the pane.

Figure 31.25 `SplitPane` provides methods to specify the properties of a split pane and for manipulating the components in a split pane.

Listing 31.11 gives an example that uses radio buttons to let the user select a country and displays the country's flag and description in separate sides, as shown in Figure 31.26. The description of the currently selected layout manager is displayed in a text area. The radio buttons, buttons, and text area are placed in two split panes.

Listing 31.11 `SplitPaneDemo.java`

```java
import javafx.application.Application;
import javafx.geometry.Orientation;
import javafx.scene.Scene;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.SplitPane;
import javafx.scene.control.TextArea;
import javafx.scene.control.ToggleGroup;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class SplitPaneDemo extends Application {
  private Image usImage = new Image(
    "http://www.cs.armstrong.edu/liang/common/image/us.gif");
  private Image ukImage = new Image(
    "http://www.cs.armstrong.edu/liang/common/image/uk.gif");
  private Image caImage = new Image(
    "http://www.cs.armstrong.edu/liang/common/image/ca.gif");
  private String usDescription = "Description for US ...";
  private String ukDescription = "Description for UK ...";
  private String caDescription = "Description for CA ...";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    VBox vBox = new VBox(10);
    RadioButton rbUS = new RadioButton("US");
    RadioButton rbUK = new RadioButton("UK");
    RadioButton rbCA = new RadioButton("CA");
    vBox.getChildren().addAll(rbUS, rbUK, rbCA);

    SplitPane content = new SplitPane();
    content.setOrientation(Orientation.VERTICAL);
    ImageView imageView = new ImageView(usImage);
    StackPane imagePane = new StackPane();
    imagePane.getChildren().add(imageView);
    TextArea taDescription = new TextArea();
    taDescription.setText(usDescription);
    content.getItems().addAll(
      imagePane, new ScrollPane(taDescription));

    SplitPane sp = new SplitPane();
    sp.getItems().addAll(vBox, content);

    Scene scene = new Scene(sp, 300, 250);
    primaryStage.setTitle("SplitPaneDemo"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window

    // Group radio buttons
    ToggleGroup group = new ToggleGroup();
    rbUS.setToggleGroup(group);
    rbUK.setToggleGroup(group);
    rbCA.setToggleGroup(group);

    rbUS.setSelected(true);
    rbUS.setOnAction(e -> {
      imageView.setImage(usImage);
      taDescription.setText(usDescription);
    });

    rbUK.setOnAction(e -> {
      imageView.setImage(ukImage);
      taDescription.setText(ukDescription);
    });

    rbCA.setOnAction(e -> {
      imageView.setImage(caImage);
      taDescription.setText(caDescription);
    });
  }
}
```



Description for US ...

Description for UK ...

FIGURE 31.26 You can adjust the component size in the split panes.

The program places three radio buttons in a `VBox` (lines 28–32) and creates a vertical split pane for holding an image view and a text area (lines 34–42). Split panes can be embedded. The program creates a horizontal split pane and places the `VBox` and the vertical split pane into it (lines 44 and 45).

Adding a split pane to an existing split pane results in three split panes. The program creates two split panes (lines 34, 42) to hold a panel for radio buttons, a panel for buttons, and a scroll pane.

The program groups radio buttons (lines 53–56) and processes the action for radio buttons (lines 59–72).

Check Point

31.8.1 How do you create a horizontal `SplitPane`? How do you create a vertical `SplitPane`?

31.8.2 How do you add items into a `SplitPane`? Can you add a `SplitPane` to another `SplitPane`?

31.9 TabPane

Key Point

The `TabPane` *class can be used to display multiple panes with tabs.*

`TabPane` is a useful control that provides a set of mutually exclusive tabs, as shown in Figure 31.27. You can switch between a group of tabs. Only one tab is visible at a time. A Tab can be added to a `TabPane`. Tabs in a `TabPane` can be placed in the position top, left, bottom, or right.

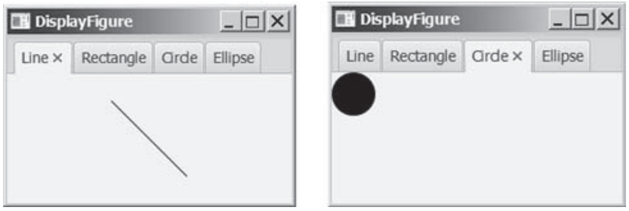
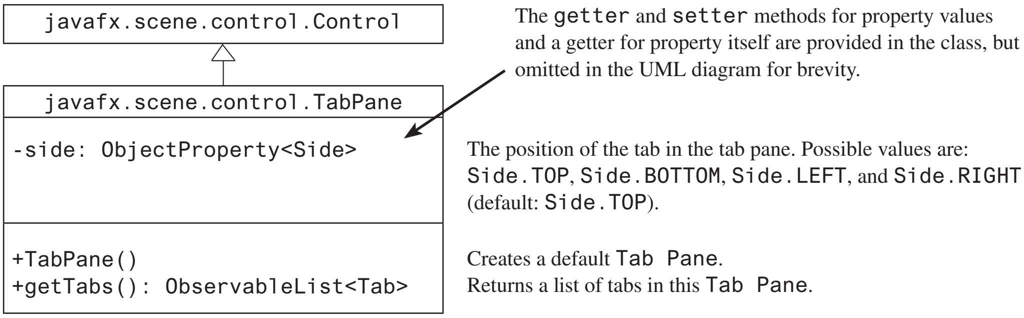

DisplayFigure
Line
Rectangle
Circle
Ellipse
DisplayFigure
Line
Rectangle
Circle
Ellipse

FIGURE 31.27 `TabPane` holds a group of tabs.

Each tab represents a single page. Tabs are defined in the `Tab` class. Tabs can contain any `Node` such as a pane, a shape, or a control. A tab can contain another pane. Therefore, you can create a multilayered tab pane. Figures 31.28 and 31.29 show the frequently used properties, constructors, and methods in `TabPane` and `Tab`.

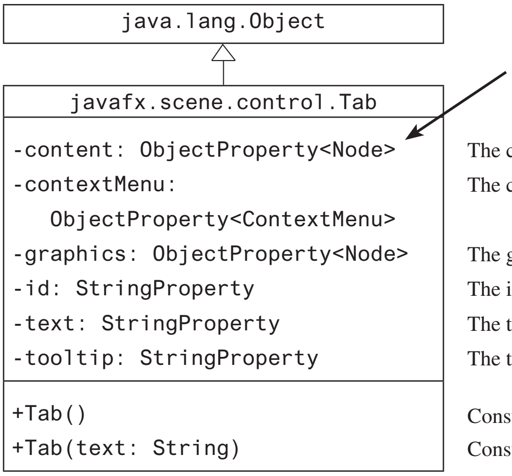

javafx.scene.control.Control
javafx.scene.control.TabPane
-side: ObjectProperty<Side>
+TabPane()
+getTabs(): ObservableList<Tab>

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The position of the tab in the tab pane. Possible values are: `Side.TOP`, `Side.BOTTOM`, `Side.LEFT`, and `Side.RIGHT` (default: `Side.TOP`).

Creates a default `Tab Pane`.
Returns a list of tabs in this `Tab Pane`.

FIGURE 31.28 `TabPane` displays and manages the tabs.


java.lang.Object
javafx.scene.control.Tab
-content: ObjectProperty<Node>
-contextMenu: ObjectProperty<ContextMenu>
-graphics: ObjectProperty<Node>
-id: StringProperty
-text: StringProperty
-tooltip: StringProperty
+Tab()
+Tab(text: String)

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The content associated with the tab.

The context menu associated with the tab.

The graphics in the tab.

The id for the tab.

The text shown in the tab.

The tooltip associated with the tab.

Constructs a default tab.

Constructs a tab with the specified string.

FIGURE 31.29 `Tab` contains a node.

Listing 31.12 gives an example that uses a tab pane with four tabs to display four types of figures: line, rectangle, rounded rectangle, and oval. You can select a figure to display by clicking the corresponding tab, as shown in Figure 31.27.

LISTING 31.12 TabPaneDemo.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Tab;
import javafx.scene.control.TabPane;
import javafx.scene.layout.StackPane;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Ellipse;
import javafx.scene.shape.Line;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class TabPaneDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    TabPane tabPane = new TabPane();
    Tab tab1 = new Tab("Line");
    StackPane pane1 = new StackPane();
    pane1.getChildren().add(new Line(10, 10, 80, 80));
    tab1.setContent(pane1);
    Tab tab2 = new Tab("Rectangle");
    tab2.setContent(new Rectangle(10, 10, 200, 200));
    Tab tab3 = new Tab("Circle");
    tab3.setContent(new Circle(50, 50, 20));
    Tab tab4 = new Tab("Ellipse");
    tab4.setContent(new Ellipse(10, 10, 100, 80));
    tabPane.getTabs().addAll(tab1, tab2, tab3, tab4);

    Scene scene = new Scene(tabPane, 300, 250);
    primaryStage.setTitle("DisplayFigure"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window
  }
}
```

The program creates a tab pane (line 15) and four tabs (lines 16, 20, 22, and 24). A stack pane is created to hold a line (line 18) and placed into `tab1` (line 19). A rectangle, circle, and oval are created and placed into `tab2`, `tab3`, and `tab4`. Note the line is centered in `tab1` because it is placed in a stack pane. The other shapes are directly placed into the tab. They are displayed at the upper left corner of the tab.

By default, the tabs are placed at the top of the tab pane. You can use the `setSide` method to change its location.

Check Point

31.9.1 How do you create a tab pane? How do you create a tab? How do you add a tab to a tab pane?

31.9.2 How do you place the tabs on the left of the tab pane?

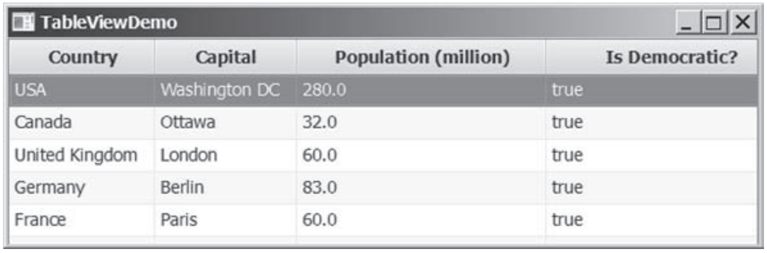
31.9.3 Can a tab have a text as well as an image? Write the code to set an image for `tab1` in Listing 31.12.

31.10 TableView

Key Point

You can display tables using the `TableView` class.

`TableView` is a control that displays data in rows and columns in a two-dimensional grid, as shown in Figure 31.30.



Country	Capital	Population (million)	Is Democratic?
USA	Washington DC	280.0	true
Canada	Ottawa	32.0	true
United Kingdom	London	60.0	true
Germany	Berlin	83.0	true
France	Paris	60.0	true

FIGURE 31.30 `TableView` displays data in a table.

`TableView`, `TableColumn`, and `TableCell` are used to display and manipulate a table. `TableView` displays a table. `TableColumn` defines the columns in a table. `TableCell` represents a cell in the table. Creating a `TableView` is a multistep process. First, you need to create an instance of `TableView` and associate data with the `TableView`. Second, you need to create columns using the `TableColumn` class and set a column cell value factory to specify how to populate all cells within a single `TableColumn`.

Listing 31.13 gives a simple example to demonstrate using `TableView` and `TableColumn`. A sample run of the program is shown in Figure 31.31.

LISTING 31.13 TableViewDemo.java

```java
import javafx.application.Application;
import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;
```

```

public class TableViewDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    TableView<Country> tableView = new TableView<>();
    ObservableList<Country> data =
      FXCollections.observableArrayList(
        new Country("USA", "Washington DC", 280, true),
        new Country("Canada", "Ottawa", 32, true),
        new Country("United Kingdom", "London", 60, true),
        new Country("Germany", "Berlin", 83, true),
        new Country("France", "Paris", 60, true));
    tableView.setItems(data);

    TableColumn countryColumn = new TableColumn("Country");
    countryColumn.setMinWidth(100);
    countryColumn.setCellValueFactory(
      new PropertyValueFactory<Country, String>("country"));

    TableColumn capitalColumn = new TableColumn("Capital");
    capitalColumn.setMinWidth(100);
    capitalColumn.setCellValueFactory(
      new PropertyValueFactory<Country, String>("capital"));

    TableColumn populationColumn =
      new TableColumn("Population (million)");
    populationColumn.setMinWidth(200);
    populationColumn.setCellValueFactory(
      new PropertyValueFactory<Country, Double>("population"));

    TableColumn democraticColumn =
      new TableColumn("Is Democratic?");
    democraticColumn.setMinWidth(200);
    democraticColumn.setCellValueFactory(
      new PropertyValueFactory<Country, Boolean>("democratic"));

    tableView.getColumns().addAll(countryColumn, capitalColumn,
      populationColumn, democraticColumn);

    Pane pane = new Pane();
    pane.getChildren().add(tableView);
    Scene scene = new Scene(pane, 300, 250);
    primaryStage.setTitle("TableViewDemo"); // Set the window  title
    primaryStage.setScene(scene); // Place the scene in t he window
    primaryStage.show(); // Display the window
  }

  public static class Country {
    private final SimpleStringProperty country;
    private final SimpleStringProperty capital;
    private final SimpleDoubleProperty population;
    private final SimpleBooleanProperty democratic;

    private Country(String country, String capital,
       double population, boolean democratic) {
      this.country = new SimpleStringProperty(country);
      this.capital = new SimpleStringProperty(capital);
      this.population = new SimpleDoubleProperty(population);
      this.democratic = new SimpleBooleanProperty(democratic);
    }
73
74       public String getCountry() {
75         return country.get();
76       }
77
78       public void setCountry(String country) {
79         this.country.set(country);
80       }
81
82       public String getCapital() {
83         return capital.get();
84       }
85
86       public void setCapital(String capital) {
87         this.capital.set(capital);
88       }
89
90       public double getPopulation() {
91         return population.get();
92       }
93
94       public void setPopulation(double population) {
95         this.population.set(population);
96       }
97
98       public boolean isDemocratic() {
99         return democratic.get();
100      }
101
102      public void setDemocratic(boolean democratic) {
103        this.democratic.set(democratic);
104      }
105    }
106  }
```

The program creates a `TableView` (line 17). The `TableView` class is a generic class whose concrete type is Country. Therefore, this `TableView` is for displaying Country. The table data is an `ObservableList<Country>`. The program creates the list (lines 18–24) and associates the list with the `TableView` (line 25).

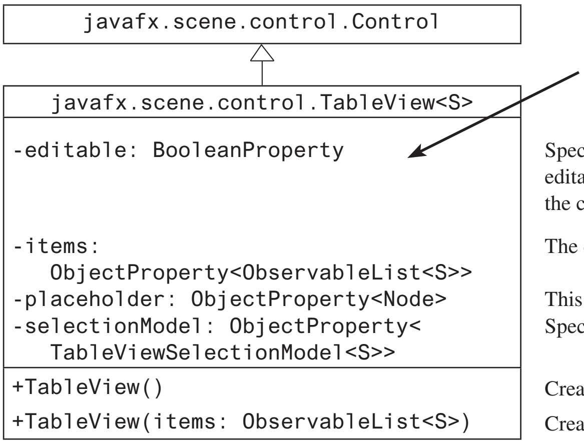
The program creates a `TableColumn` for each column in the table (lines 27–47). A `PropertyValueFactory` object is created and set for each column (line 30). This object is used to populate the data in the column. The `PropertyValueFactory<S, T>` class is a generic class. `S` is for the class displayed in the `TableView` and `T` is the class for the values in the column. The `PropertyValueFactory` object associates a property in class `S` with a column.

When you create a table in a JavaFX application, it is a best practice to define the data model in a class. The `Country` class defines the data for `TableView`. Each property in the class defines a column in the table. This property should be defined as binding property with the getter and setter methods for the value.

The program adds the columns into the `TableView` (lines 49 and 50), adds the `TableView` in a pane (line 53), and places the pane in the scene (line 54). Note line 31 can be simplified using the following code:

```
new PropertyValueFactory<>("country");
```

From this example, you see how to display data in a table using the `TableView` and `TableColumn` classes. The frequently used properties and methods for the `TableView` and `TableColumn` classes are given in Figures 31.32 and 31.33.


javafx.scene.control.Control
javafx.scene.control.TableView<S>
-editable: BooleanProperty
-items: ObjectProperty<ObservableList<S>>
-placeholder: ObjectProperty<Node>
-selectionModel: ObjectProperty< TableViewSelectionModel<S>>
+TableView()
+TableView(items: ObservableList<S>)

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Specifies whether this `TableView` is editable. For a cell to be editable, `TableView`, `TableColumn`, and `TableCell` for the cell should all be true.

The data model for the `TableView`.

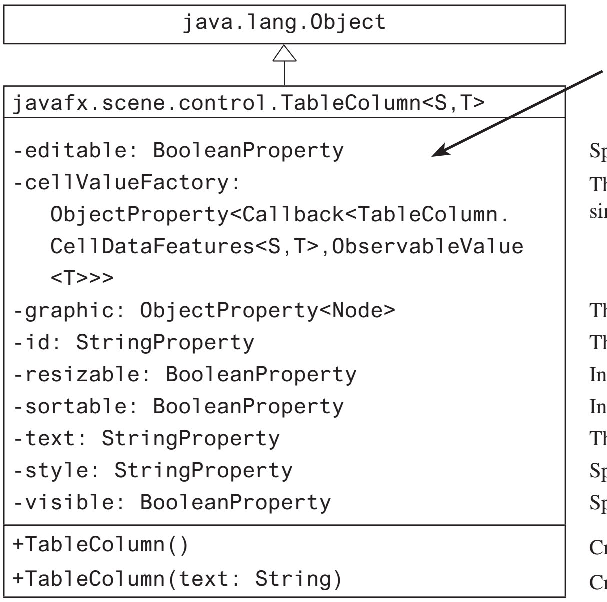
This Node is shown when table has no contents.

Specifies single or multiple selections.

Creates a default `TableView` with no content.

Creates a default `TableView` with the specified content.

FIGURE 31.31 `TableView` displays a table.


java.lang.Object
javafx.scene.control.TableColumn<S,T>
-editable: BooleanProperty
-cellValueFactory: ObjectProperty<Callback<TableColumn. CellDataFeatures<S,T>,ObservableValue <T>>>
-graphic: ObjectProperty<Node>
-id: StringProperty
-resizable: BooleanProperty
-sortable: BooleanProperty
-text: StringProperty
-style: StringProperty
-visible: BooleanProperty
+TableColumn()
+TableColumn(text: String)

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Specifies whether this `TableColumn` allows editing.

The cell value factory to specify how to populate all cells within a single column.

The graphic for this `TableColumn`.

The id for this `TableColumn`.

Indicates whether the column is resizable.

Indicates whether the column is sortable.

The text in the table column header.

Specifies the CSS style for the column.

Specifies whether the column is visible (default: true).

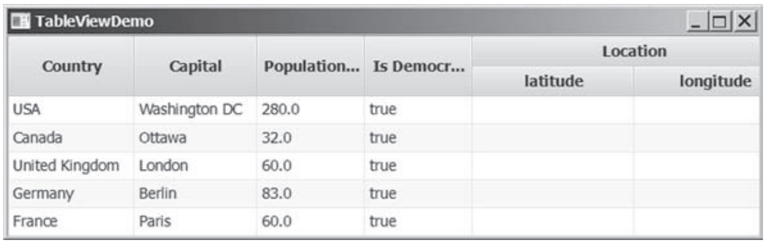
Creates a default `TableColumn`.

Creates a TableView with the specified header text.

FIGURE 31.32 `TableColumn` defines a column in the TableView.

You can create nested columns. For example, the following code creates two subcolumns under Location, as shown in Figures 31.33.

```
TableColumn locationColumn = new TableColumn("Location");
locationColumn.getColumns().addAll(new TableColumn("latitude"),
  new TableColumn("longitude"));
```


TableViewDemo

Country	Capital	Population...	Is Democr...	Location	
				latitude	longitude
USA	Washington DC	280.0	true		
Canada	Ottawa	32.0	true		
United Kingdom	London	60.0	true		
Germany	Berlin	83.0	true		
France	Paris	60.0	true		

FIGURE 31.33 You can add subcolumns in a column.

The `TableView` data model is an observable list. When data is changed, the change is automatically shown in the table. Listing 31.14 gives an example that lets the user add new rows to the table.

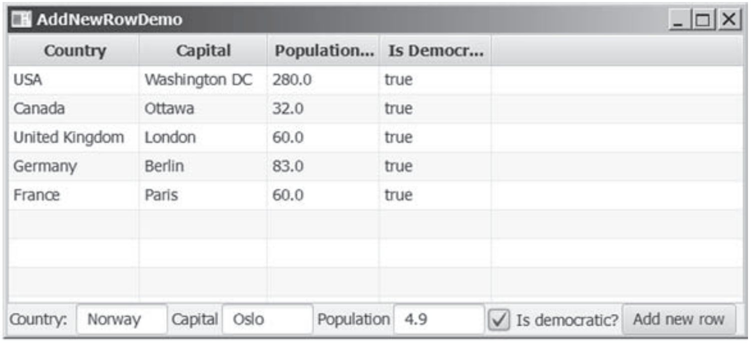
LISTING 31.14 AddNewRowDemo.java

```
import javafx.application.Application;
import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

public class AddNewRowDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    TableView<Country> tableView = new TableView<>();
    ObservableList<Country> data =
      FXCollections.observableArrayList(
        new Country("USA", "Washington DC", 280, true),
        new Country("Canada", "Ottawa", 32, true),
        new Country("United Kingdom", "London", 60, true),
        new Country("Germany", "Berlin", 83, true),
        new Country("France", "Paris", 60, true));
    tableView.setItems(data);

    TableColumn countryColumn = new TableColumn("Country");
    countryColumn.setMinWidth(100);
    countryColumn.setCellValueFactory(
      new PropertyValueFactory<Country, String>("country"));

    TableColumn capitalColumn = new TableColumn("Capital");
    capitalColumn.setMinWidth(100);
    capitalColumn.setCellValueFactory(
      new PropertyValueFactory<Country, String>("capital"));

    TableColumn populationColumn =
      new TableColumn("Population (million)");
    populationColumn.setMinWidth(100);
    populationColumn.setCellValueFactory(
      new PropertyValueFactory<Country, Double>("population"));

    TableColumn democraticColumn =
      new TableColumn("Is Democratic?");
    democraticColumn.setMinWidth(100);
    democraticColumn.setCellValueFactory(
      new PropertyValueFactory<Country, Boolean>("democratic"));

    tableView.getColumns().addAll(countryColumn, capitalColumn,
      populationColumn, democraticColumn);

    FlowPane flowPane = new FlowPane(3, 3);
    TextField tfCountry = new TextField();
    TextField tfCapital = new TextField();
    TextField tfPopulation = new TextField();
    CheckBox chkDemocratic = new CheckBox("Is democratic?");
    Button btAddRow = new Button("Add new row");
    tfCountry.setPrefColumnCount(5);
    tfCapital.setPrefColumnCount(5);
    tfPopulation.setPrefColumnCount(5);
    flowPane.getChildren().addAll(new Label("Country: "),
      tfCountry, new Label("Capital"), tfCapital,
      new Label("Population"), tfPopulation, chkDemocratic,
      btAddRow);

    btAddRow.setOnAction(e -> {
      data.add(new Country(tfCountry.getText(), tfCapital.getText(),
        Double.parseDouble(tfPopulation.getText()),
        chkDemocratic.isSelected()));
      tfCountry.clear();
      tfCapital.clear();
      tfPopulation.clear();
    });

    BorderPane pane = new BorderPane();
    pane.setCenter(tableView);
    pane.setBottom(flowPane);

    Scene scene = new Scene(pane, 500, 250);
    primaryStage.setTitle("AddNewRowDemo"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window
  }

  public static class Country {
    private final SimpleStringProperty country;
    private final SimpleStringProperty capital;
    private final SimpleDoubleProperty population;
    private final SimpleBooleanProperty democratic;

    private Country(String country, String capital,
      double population, boolean democratic) {
    this.country = new SimpleStringProperty(country);
    this.capital = new SimpleStringProperty(capital);
    this.population = new SimpleDoubleProperty(population);
    this.democratic = new SimpleBooleanProperty(democratic);
  }

  public String getCountry() {
    return country.get();
  }

  public void setCountry(String country) {
    this.country.set(country);
  }

  public String getCapital() {
    return capital.get();
  }
```

```
115
116       public void setCapital(String capital) {
117         this.capital.set(capital);
118       }
119
120       public double getPopulation() {
121         return population.get();
122       }
123
124       public void setPopulation(double population) {
125         this.population.set(population);
126       }
127
128       public boolean isDemocratic() {
129         return democratic.get();
130       }
131
132       public void setDemocratic(boolean democratic) {
133         this.democratic.set(democratic);
134       }
135     }
136   }
```

The program is the same in Listing 31.13 except that a new code is added to let the user enter a new row (lines 57–82). The user enters the new row from the text fields and a check box and presses the *Add New Row* button to add a new row to the data. Since data is an observable list, the change in data is automatically updated in the table.

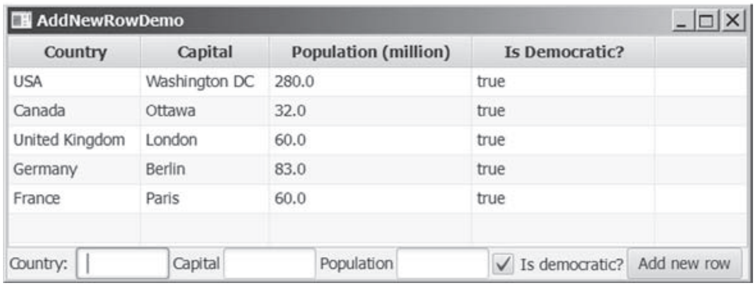
As shown in Figure 31.34a, a new country information is entered in the text fields. After clicking the *Add New Row button*, the new country is displayed in the table view.



Country	Capital	Population...	Is Democr...
USA	Washington DC	280.0	true
Canada	Ottawa	32.0	true
United Kingdom	London	60.0	true
Germany	Berlin	83.0	true
France	Paris	60.0	true

Country: Norway Capital Oslo Population 4.9 ☑ Is democratic?

(a)



Country	Capital	Population (million)	Is Democratic?
USA	Washington DC	280.0	true
Canada	Ottawa	32.0	true
United Kingdom	London	60.0	true
Germany	Berlin	83.0	true
France	Paris	60.0	true

Country: Capital Population ☑ Is democratic?

(b)

FIGURE 31.34 Change in the table data model is automatically displayed in the table view.

`TableView` not only displays data, but also allows data to be edited. To enable data editing in the table, write the code as follows:

1. Set the `TableView`'s `editable` to true.
2. Set the column's cell factory to a text field table cell.
3. Implement the column's `setOnEditCommit` method to assign the edited value to the data model.

   Here is the example of enabling editing for the `countryColumn`.

```
tableView.setEditable(true);
countryColumn.setCellFactory(TextFieldTableCell.forTableColumn());
countryColumn.setOnEditCommit(
  new EventHandler<CellEditEvent<Country, String>>() {
    @Override
    public void handle(CellEditEvent<Country, String> t) {
      t.getTableView().getItems().get(
        t.getTablePosition().getRow())
      .setCountry(t.getNewValue());
    }
  }
);
```

Check Point

31.10.1 How do you create a table view? How do you create a table column? How do you add a table column to a table view?

31.10.2 What is the data type for a `TableView`'s data model? How do you associate a data model with a `TableView`?

31.10.3 How do you set a cell value factory for a `TableColumn`?

31.10.4 How do you set an image in a table column header?

31.11 Developing JavaFX Programs Using FXML

Key Point

You can create JavaFX user interfaces using FXML scripts.

There are two ways to develop JavaFX applications. One way is to write everything in Java source code as you have done so far. The other way is to use FXML. FXML is an XML-based script language for describing the user interface. Using FXML enables you to separate user interface from the logic of the Java code. JavaFX Scene Builder is a visual design tool for creating the user interface without manually writing the FXML script. You drag and drop the UI components to the content pane and set properties for the components in the Inspector. The Scene Builder automatically generates the FXML scripts for the user interface. This section demonstrates how to use the Scene Builder to create JavaFX applications.

NOTE

It is important that you first learn how to write the JavaFX code without using FXML to grasp the fundamentals of JavaFX programming before learning FXML. Once you understand the basics of JavaFX, it is easy to create JavaFX programs using FXML. For this reason, FXML is introduced after you have learned the basics of JavaFX programming.

31.11.1 Installing JavaFX Scene Builder

You can use the JavaFX Scene Builder standalone or with an IDE such as NetBeans or Eclipse. This section demonstrates using the JavaFX Scene Builder with NetBeans. You can download the latest version of Scene Builder from http://gluonhq.com/open-source/scene-builder/.

31.11.2 Creating a JavaFX FXML Project

To use JavaFX FXML, you need to create a JavaFX FXML in NetBeans. Here are the steps of creating a JavaFX FXML project:

1. Choose *File*, *New Project* to display the New Project dialog box, as shown in Figure 31.35.
2. Choose *JavaFX* in the Categories and *JavaFX FXML Application* in the Projects. Click *Next* to display the New JavaFX Application dialog box, as shown in Figure 31.36.
3. Enter Calculator as the project name and click *Finish* to create the project. You will see the project created as shown in Figure 31.37.

Three files, `Calculator.java`, `FXMLDocument.fxml`, and `FXMLDocumentController.java`, are created in the project. Their source codes are shown in Listings 31.15, 31.16, and 31.17. From the perspective of the MVC architecture, these three files correspond to model, view, and controller. You can define data model in the `Calculator.java` class. The .fxml file describes the user interface. The controller file defines the actions for processing the events for the user interface.

LISTING 31.15 Calculator.java

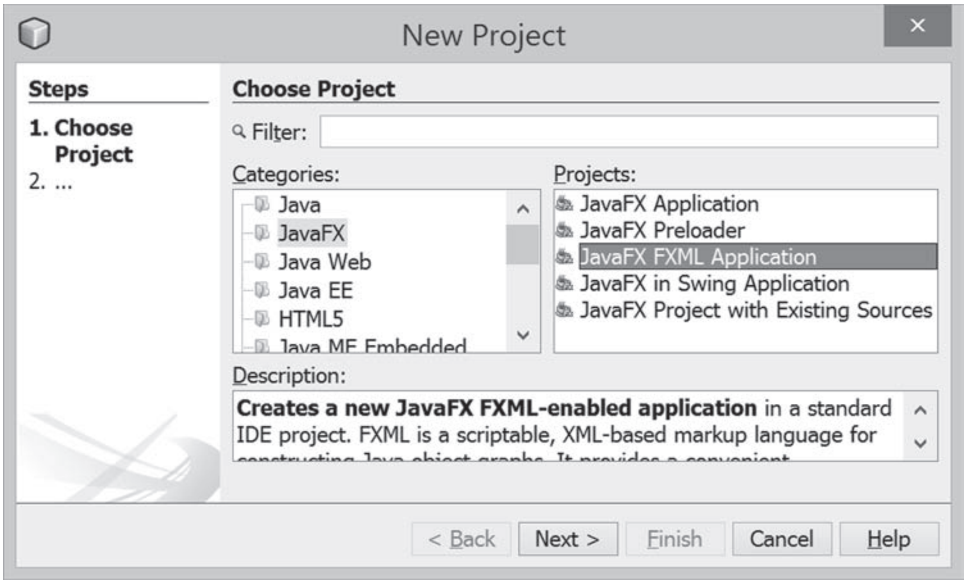
```java
package calculator;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Calculator extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    Parent root =
      FXMLLoader.load(getClass().getResource("FXMLDocument.fxml"));
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.show();
  }

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```

LISTING 31.16 FXMLDocument.fxml

```xml
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
            xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="calculator.FXMLDocumentController">
    <children>
        <Button layoutX="126" layoutY="90" text="Click Me!"
                onAction="#handleButtonAction" fx:id="button" />
        <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69"
               fx:id="label" />
     </children>
</AnchorPane>
```

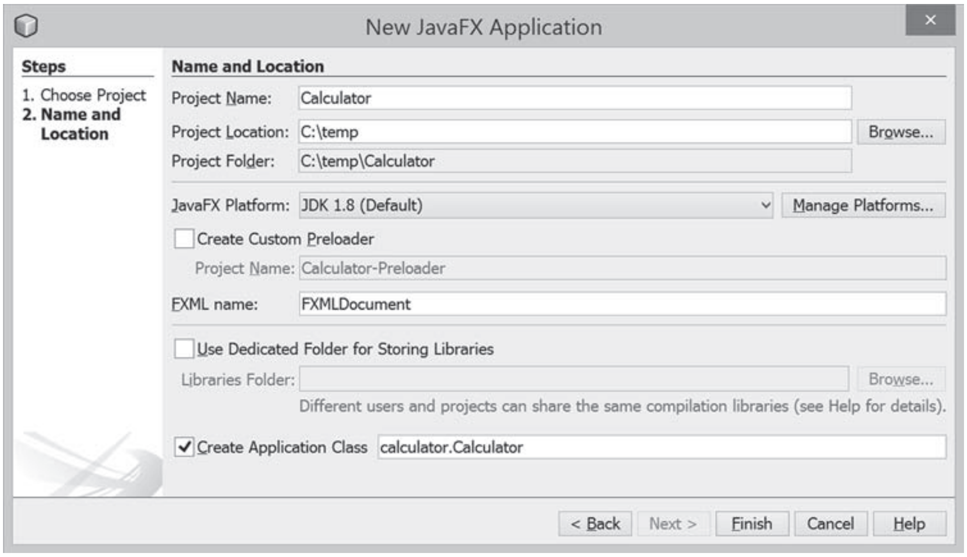
LISTING 31.17 FXMLDocumentController.java

```
package calculator;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Label;

public class FXMLDocumentController implements Initializable {
  @FXML
  private Label label;

  @FXML
  private void handleButtonAction(ActionEvent event) {
    System.out.println("You clicked me!");
    label.setText("Hello World!");
  }

  @Override
  public void initialize(URL url, ResourceBundle rb) {
    // TODO
  }
}
```


New Project
Steps
1. Choose Project
2. ...
Choose Project
Filter:
Categories:
Java
JavaFX
Java Web
Java EE
HTML5
Java ME Embedded
Projects:
JavaFX Application
JavaFX Preloader
JavaFX FXML Application
JavaFX in Swing Application
JavaFX Project with Existing Sources
Description:
< Back
Next >
Finish
Cancel
Help

Creates a new JavaFX FXML-enabled application in a standard IDE project. FXML is a scriptable, XML-based markup language for

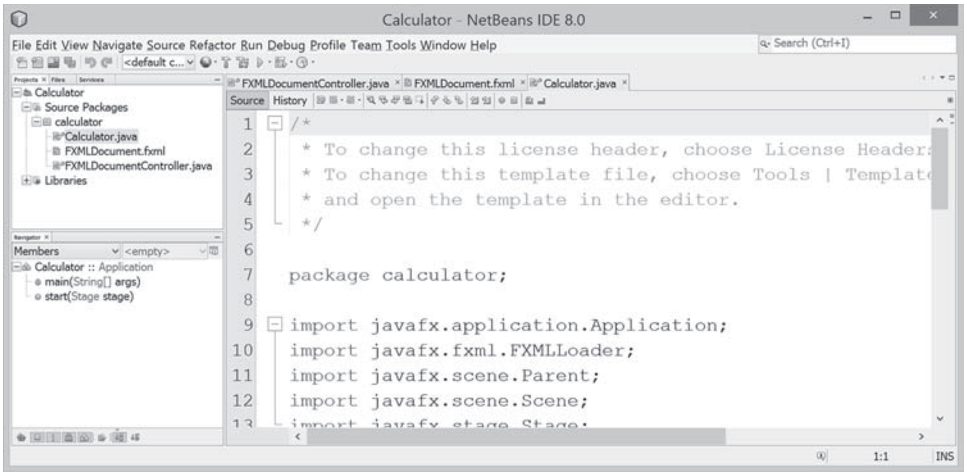
FIGURE 31.35 You can choose JavaFX in the Categories and JavaFX FXML Application in the Project to create a FXML project.


New JavaFX Application
Steps
1. Choose Project
2. Name and Location
Name and Location

Project Name: Calculator
Project Location: C:\temp
Project Folder: C:\temp\Calculator
JavaFX Platform: JDK 1.8 (Default)
☐ Create Custom Preloader
Project Name: Calculator-Preloader
FXML name: FXMLDocument
☐ Use Dedicated Folder for Storing Libraries
Libraries Folder:
Different users and projects can share the same compilation libraries (see Help for details).
☑ Create Application Class calculator.Calculator

Browse...
Manage Platforms...
Browse...
< Back
Next >
Finish
Cancel
Help

Figure 31.36 You can enter project information in the New JavaFX Application dialog.

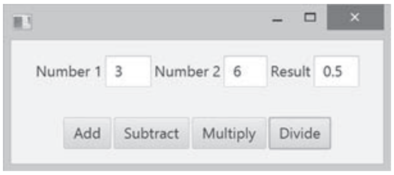

Calculator - NetBeans IDE 8.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default c...
Projects
Files
Services
Calculator
Source Packages
calculator
Calculator.java
FXMLDocument.fxml
FXMLDocumentController.java
Libraries
Navigator
Members
<empty>
Calculator :: Application
main(String[] args)
start(Stage stage)
FXMLDocumentController.java
FXMLDocument.fxml
Calculator.java
Source
History
1:1
INS

```
/*
 * To change this license header, choose License Header
 * To change this template file, choose Tools | Templat
 * and open the template in the editor.
 */

package calculator;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;
```

Figure 31.37 A FXML project is created.

31.11.3 Creating User Interfaces

We now turn our attention to developing a simple calculator as shown in Figure 31.38. The Calculator program enables the user to enter numbers and perform addition, subtraction, multiplication, and division.


Number 1 3 Number 2 6 Result 0.5

Add
Subtract
Multiply
Divide

Figure 31.38 The application performs arithmetic operations.

When you create a JavaFX FXML project, NetBeans creates a default .fxml file that contains the contents for a simple sample user interface. To view the user interface, double-click the .fxml file to open the Scene Builder, as shown in Figure 31.39. Note NetBeans can automatically detect the Scene Builder after it is installed on your machine.

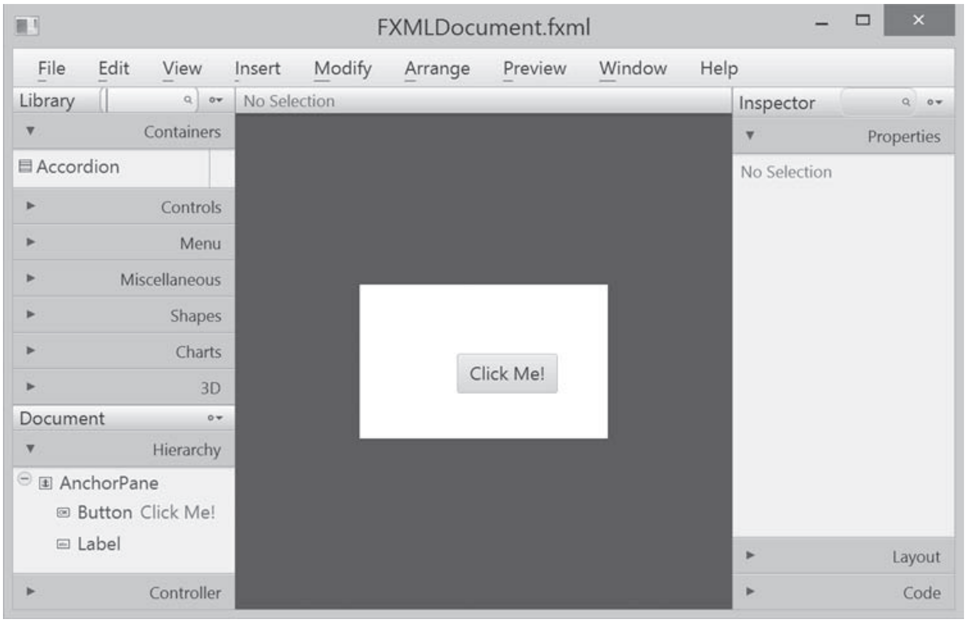

FXMLDocument.fxml
File
Edit
View
Insert
Modify
Arrange
Preview
Window
Help
Library
No Selection
Inspector
Containers
Properties
Accordion
No Selection
Controls
Menu
Miscellaneous
Shapes
Charts
3D
Click Me!
Document
Hierarchy
AnchorPane
Button Click Me!
Label
Controller
Layout
Code

FIGURE 31.39 Double-click the .fxml file to open the Scene Builder.

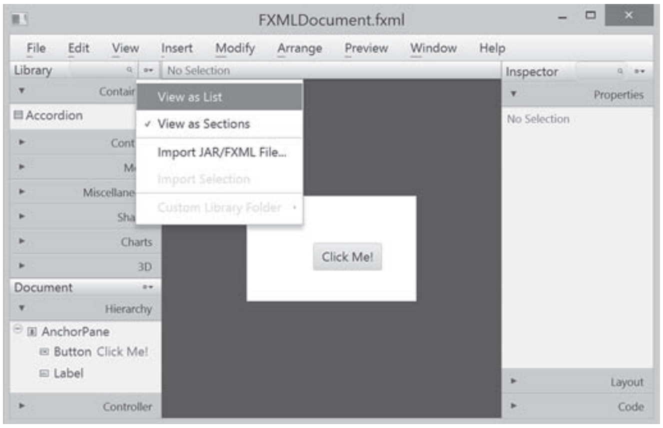

FXMLDocument.fxml
File
Edit
View
Insert
Modify
Arrange
Preview
Window
Help
Library
No Selection
Inspector
Contair
Properties
Accordion
No Selection
View as List
✓ View as Sections
Import JAR/FXML File...
Import Selection
Custom Library Folder
Cont
M
Miscellane
Sha
Charts
3D
Click Me!
Document
Hierarchy
AnchorPane
Button Click Me!
Label
Controller
Layout
Code

FIGURE 31.40 You can open the Library pane by clicking the Library icon and choose View as List.

To start a new user interface, delete the default user interface in the .fxml file from the content pane, as shown in Figure 31.41. Here are the steps to create a new user interface:

1. (Optional) On some systems, the components in the Library pane are not visible by sections. Click the Library icon to open the context menu as shown in Figure 31.40 and choose *View as List*.
2. Drag a `BorderPane` into the user interface and drag an `HBox` to the center of the `BorderPane` and another `HBox` to the bottom of the `BorderPane`. Set the alignment of both `HBox` to `CENTER` as shown in Figure 31.42. Set the `Spacing` property in the Layout section of the Inspector to 5. When you select a component in the visual layout, the properties of the component are displayed in the Inspector pane, where you can set the properties.

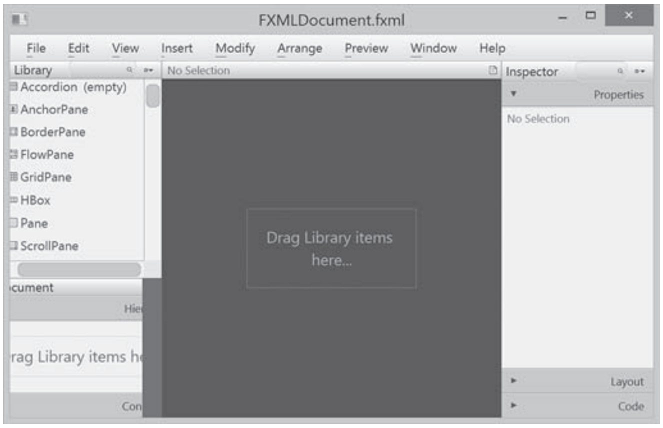

FXMLDocument.fxml
File
Edit
View
Insert
Modify
Arrange
Preview
Window
Help
Library
Accordion (empty)
AnchorPane
BorderPane
FlowPane
GridPane
HBox
Pane
ScrollPane
No Selection
Drag Library items here...
Inspector
Properties
No Selection
Layout
Code

FIGURE 31.41 The UI is empty after deleting the default button in the pane.

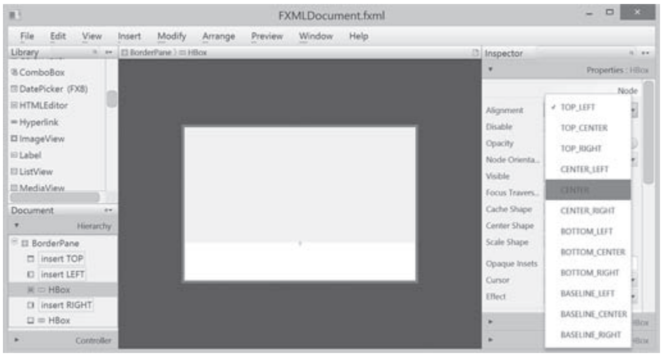

FXMLDocument.fxml
File
Edit
View
Insert
Modify
Arrange
Preview
Window
Help
Library
ComboBox
DatePicker (FX8)
HTMLEditor
Hyperlink
ImageView
Label
ListView
MediaView
Document
Hierarchy
BorderPane
insert TOP
insert LEFT
HBox
insert RIGHT
HBox
Controller
BorderPane > HBox
Inspector
Properties : HBox
Node
Alignment
Disable
Opacity
Node Orienta...
Visible
Focus Travers...
Cache Shape
Center Shape
Scale Shape
Opaque Insets
Cursor
Effect
TOP_LEFT
TOP_CENTER
TOP_RIGHT
CENTER_LEFT
CENTER
CENTER_RIGHT
BOTTOM_LEFT
BOTTOM_CENTER
BOTTOM_RIGHT
BASELINE_LEFT
BASELINE_CENTER
BASELINE_RIGHT

FIGURE 31.42 A BorderPane is dropped to the UI and an HBox is placed at the bottom of the BorderPane.

3. Drag and drop a `Label`, a `TextField`, a `Label`, a `TextField`, a `Label`, and a `TextField` and change the label's text to Number 1, Number 2, and Result, as shown in Figure 31.43. Set the Pref Column Count property for each text field to 2 in the Layout section of the Inspector. In the Code section of the Inspector, set the id for the text fields to `tfNumber1`, `tfNumber2`, and `tfResult`, as shown in Figure 31.44. These ids are useful to reference the text fields and obtain their values in the controller.

4. Drag and drop four `Buttons` to the second `HBox` and set their text property to Add, Subtract, Multiply, and Divide, as shown in Figure 31.45.

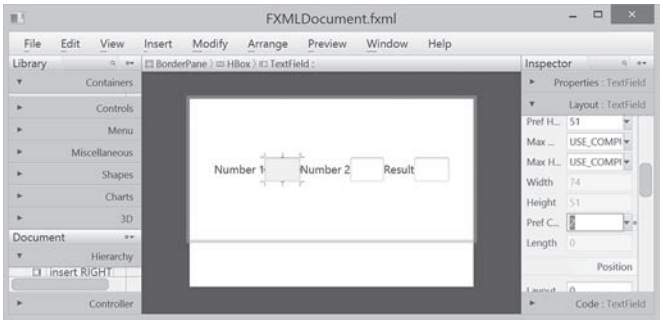
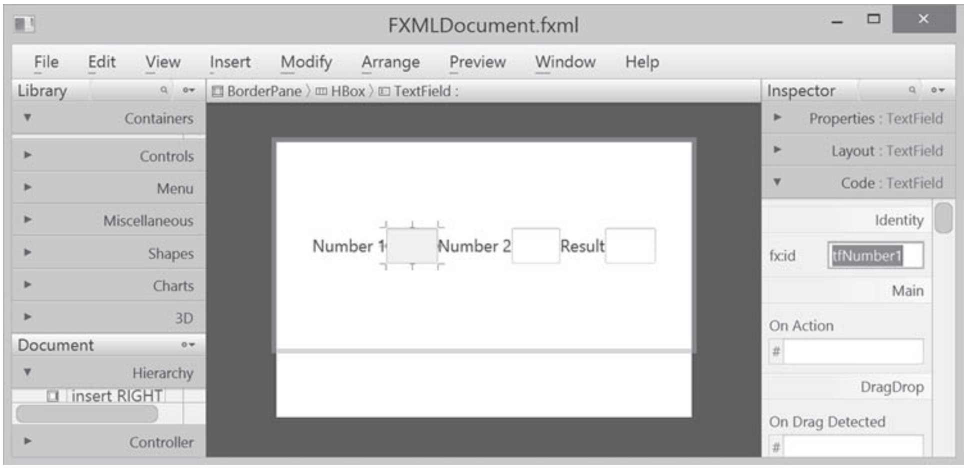
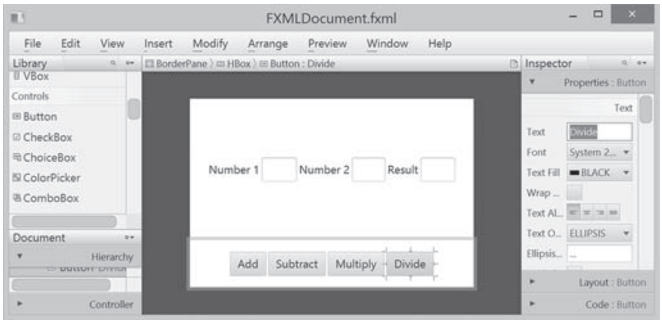

FXMLDocument.fxml
File
Edit
View
Insert
Modify
Arrange
Preview
Window
Help
Library
Containers
Controls
Menu
Miscellaneous
Shapes
Charts
3D
Document
Hierarchy
insert RIGHT
Controller
BorderPane > HBox > TextField
Number 1
Number 2
Result
Inspector
Properties : TextField
Layout : TextField
Pref H... 51
Max ... USE_COMPU
Max H... USE_COMPU
Width 74
Height 51
Pref C...
Length 0
Position
Code : TextField

FIGURE 31.43 The labels and text fields are dropped to the UI.


FXMLDocument.fxml
File
Edit
View
Insert
Modify
Arrange
Preview
Window
Help
Library
Containers
Controls
Menu
Miscellaneous
Shapes
Charts
3D
Document
Hierarchy
insert RIGHT
Controller
BorderPane 〉 HBox 〉 TextField :
Number 1
Number 2
Result
Inspector
Properties : TextField
Layout : TextField
Code : TextField
Identity
Main
On Action
DragDrop
On Drag Detected

fx:id tfNumber1

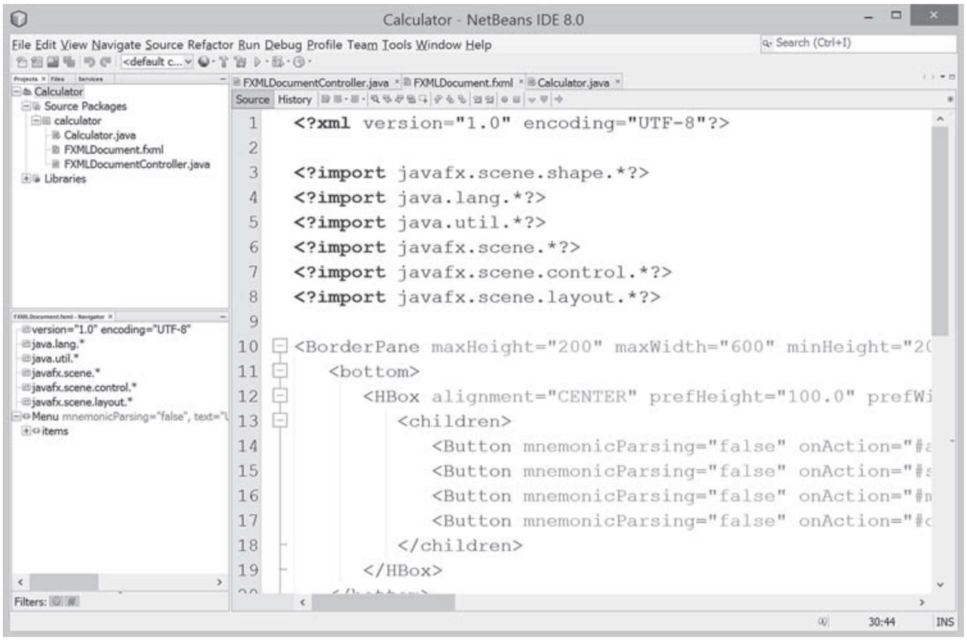
FIGURE 31.44 Set the appropriate id for the text fields.


FXMLDocument.fxml
File
Edit
View
Insert
Modify
Arrange
Preview
Window
Help
Library
VBox
Controls
Button
CheckBox
ChoiceBox
ColorPicker
ComboBox
Document
Hierarchy
Controller
BorderPane 〉 HBox 〉 Button : Divide
Number 1
Number 2
Result
Add
Subtract
Multiply
Divide
Inspector
Properties : Button
Text
Layout : Button
Code : Button

Text Divide
Font System 2...
Text Fill BLACK
Wrap ...
Text Al...
Text O... ELLIPSIS
Ellipsis... ...

FIGURE 31.45 The buttons are dropped to the HBox.

After you create and make changes to the user interface in the content pane, you need to save the changes by choosing *File*, *Save* from the menu bar in the Scene Builder. The .fxml file is updated and synchronized with the changes in the content pane. You can view the contents in the .fxml file from NetBeans, as shown in Figure 31.46.


Calculator - NetBeans IDE 8.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default c...
FXMLDocumentController.java
FXMLDocument.fxml
Calculator.java
Source
History
Calculator
Source Packages
calculator
Calculator.java
FXMLDocument.fxml
FXMLDocumentController.java
Libraries
version="1.0" encoding="UTF-8"
java.lang.*
java.util.*
javafx.scene.*
javafx.scene.control.*
javafx.scene.layout.*
Menu mnemonicParsing="false", text="
items
Filters:
30:44
INS

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.shape.*?>
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<BorderPane maxHeight="200" maxWidth="600" minHeight="2
   <bottom>
      <HBox alignment="CENTER" prefHeight="100.0" prefWi
         <children>
            <Button mnemonicParsing="false" onAction="#a
            <Button mnemonicParsing="false" onAction="#s
            <Button mnemonicParsing="false" onAction="#m
            <Button mnemonicParsing="false" onAction="#d
         </children>
      </HBox>
```

FIGURE 31.46 You can view the contents of the FXML file.

31.11.4 Handling Events in the Controller

The .fxml file describes the user interface. You write the code to implement the logic in the controller file, as shown in Listing 31.18.

LISTING 31.18 `FXMLDocumentController.java`

```
 1  package calculator;
 2
 3  import javafx.event.ActionEvent;
 4  import javafx.fxml.FXML;
 5  import javafx.scene.control.TextField;
 6
 7  public class FXMLDocumentController {
 8    @FXML
 9    private TextField tfNumber1, tfNumber2, tfResult;
10
11    @FXML
12    private void addButtonAction(ActionEvent event) {
13      tfResult.setText(getResult('+') + "");
14    }
15
16    @FXML
17    private void subtractButtonAction(ActionEvent event) {
18      tfResult.setText(getResult('-') + "");
19    }
20
21    @FXML
22    private void multiplyButtonAction(ActionEvent event) {
23      tfResult.setText(getResult('*') + "");
24    }
25
26    @FXML
27    private void divideButtonAction(ActionEvent event) {
28      tfResult.setText(getResult('/') + "");
29    }
30
31    private double getResult(char op) {
32      double number1 = Double.parseDouble(tfNumber1.getText());
33      double number2 = Double.parseDouble(tfNumber2.getText());
34      switch (op) {
35        case '+':  return number1 + number2;
36        case '-':  return number1 - number2;
37        case '*':  return number1 * number2;
38        case '/':  return number1 / number2;
39      }
40      return Double.NaN;
41    }
42  }
```

The controller class declares three `TextFields`, `tfNumber1`, `tfNumber2`, and `tfResult` (line 9). The @FXML annotation denotes that these data fields are linked to the text fields in the user interface. Recall in the user interface, we set the id for the three text fields as `tfNumber1`, `tfNumber2`, and `tfResult`.

The codes for handling the events from the buttons are defined in the methods `addButtonAction`, `subtractButtonAction`, `multiplyButtonAction`, and `divideButtonAction` (lines 11–29). The @FXML annotation is used to denote that these methods will be tied to the button actions in the view.

Through the @FXML annotation, the data fields and methods in the controller are linked to the components and actions defined in the .fxml file.

31.11.5 Linking View with Controller

You can now link the actions from the components in the view with the processing methods in the controller. Here are the steps to accomplish it:

1. Add the following attribute in the <BorderPane> tag for using a controller with the view.

   ```
   fx:controller="calculator.FXMLDocumentController"
   ```

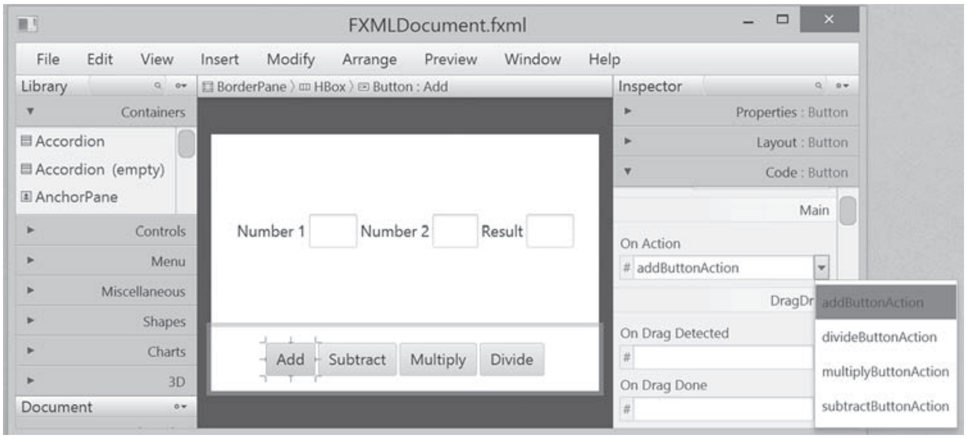
2. Double-click the .fxml file in the project to display the visual layout window. In the Inspector for the *Add* button, choose `addButtonAction` from a list of action processing methods, as shown in Figure 31.47. The complete code for the .fxml file is shown in Listing 31.19.

LISTING 31.19 FXMLDocument.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>


<BorderPane maxHeight="200" maxWidth="600" minHeight="200"
            minWidth="600" prefHeight="400.0" prefWidth="600.0"
            xmlns="http://javafx.com/javafx/8"
            xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="calculator.FXMLDocumentController">
   <bottom>
      <HBox alignment="CENTER" prefHeight="100.0" prefWidth="200.0"
           spacing="5.0" BorderPane.alignment="CENTER">
        <children>
           <Button mnemonicParsing="false"
              onAction="#addButtonAction" text="Add" />
           <Button mnemonicParsing="false"
              onAction="#subtractButtonAction" text="Subtract" />
           <Button mnemonicParsing="false"
              onAction="#multiplyButtonAction" text="Multiply" />
           <Button mnemonicParsing="false"
              onAction="#divideButtonAction" text="Divide" />
        </children>
     </HBox>
   </bottom>
   <center>
      <HBox alignment="CENTER" prefHeight="232.0" prefWidth="572.0"
           spacing="5.0" BorderPane.alignment="CENTER">
        <children>
           <Label text="Number 1" />
           <TextField fx:id="tfNumber1" prefColumnCount="2"
              prefHeight="51.0" prefWidth="74.0" />
           <Label text="Number 2" />
           <TextField fx:id="tfNumber2" prefColumnCount="2"
              prefHeight="51.0" prefWidth="70.0" />
           <Label text="Result" />
           <TextField fx:id="tfResult" prefColumnCount="2" />
        </children>
      </HBox>
   </center>
</BorderPane>
```

31.11.6 Running the Project

The code in the model is automatically generated as shown in Listing 31.15. This is the main program that loads the FXML to create the user interface in a Parent object (lines 12 and 13). The parent object is then added to the scene (line 14). The scene is set to the stage (line 15). The stage is displayed in line 16.


FXMLDocument.fxml
File
Edit
View
Insert
Modify
Arrange
Preview
Window
Help
Library
Containers
Accordion
Accordion (empty)
AnchorPane
Controls
Menu
Miscellaneous
Shapes
Charts
3D
Document
BorderPane 〉 HBox 〉 Button : Add
Number 1
Number 2
Result
Add
Subtract
Multiply
Divide
Inspector
Properties : Button
Layout : Button
Code : Button
Main
DragDr
On Drag Detected
On Drag Done
addButtonAction
divideButtonAction
multiplyButtonAction
subtractButtonAction

On Action: # addButtonAction

FIGURE 31.47 Choosing addButtonAction to generate the code for handling action for the Add button.

CHAPTER SUMMARY

1. JavaFX provides the cascading style sheets based on CSS. You can use the `getStylesheets` method to load a style sheet and use the `setStyle`, `setStyleClass`, and `setId` methods to set JavaFX CSS for nodes.
2. JavaFX provides the `QuadCurve`, `CubicCurve`, and `Path` classes for creating advanced shapes.
3. JavaFX supports coordinate transformations using translation, rotation, and scaling.
4. You can specify the pattern for a stroke, how the lines are joined in a stroke, the width of a stroke, and the type of a stroke.
5. You can create menus using the `Menu`, `MenuItem`, `CheckMenuItem`, and `RadioMenuItem` classes.
6. You can create context menus using the `ContextMenu` class.
7. The `SplitPane` can be used to display multiple panes horizontally or vertically and allows the user to adjust the sizes of the panes.
8. The `TabPane` can be used to display multiple panes with tabs for selecting panes.
9. You can create and display tables using the `TableView` and `TableColumn` classes.
10. You can create JavaFX user interfaces using FXML. FXML is XML-based script language for describing the user interface. Using FXML enables you to separate user interface from the logic of Java code.
11. JavaFX Scene Builder is a visual tool for creating the user interface without manually writing the FXML scripts.

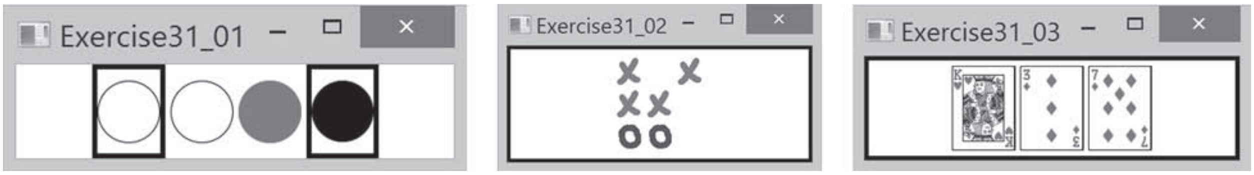
QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

Sections 31.2

31.1 (*Use JavaFX CSS*) Create a CSS style sheet that defines a class for white fill and black stroke color and an id for red stroke and green color. Write a program that displays four circles and uses the style class and id. The sample run of the program is shown in Figure 31.48a.



(a) (b) (c)

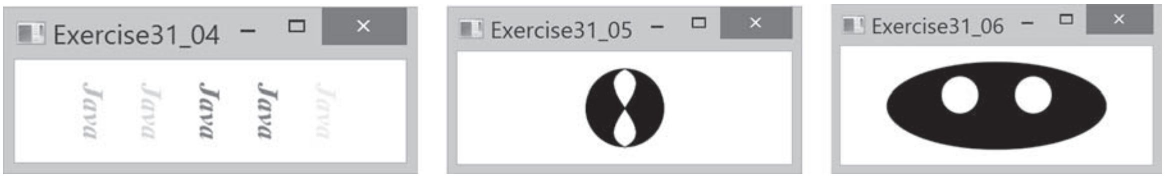
FIGURE 31.48 (a) The border and the color style for the shapes are defined in a style class. (b) Exercise 31.2 displays a tic-tac-toe board with images using style sheet for border. (c) Three cards are randomly selected.

***31.2** (*Tic-tac-toe board*) Write a program that displays a tic-tac-toe board, as shown in Figure 31.48b. A cell may be X, O, or empty. What to display at each cell is randomly decided. The X and O are images in the files `x.gif` and `o.gif`. Use the style sheet for border.

***31.3** (*Display three cards*) Write a program that displays three cards randomly selected from a deck of 52, as shown in Figure 31.48c. The card image files are named 1.png, 2.png, . . ., 52.png and stored in the `image/card` directory. All the three cards are distinct and selected randomly. Hint: You can select random cards by storing the numbers 1–52 to an array, perform a random shuffle using Section 7.2.6, and use the first three numbers in the array as the file names for the image. Use the style sheet for border.

Sections 31.3

31.4 (*Color and font*) Write a program that displays five texts vertically, as shown in Figure 31.49a. Set a random color and opacity for each text and set the font of each text to Times Roman, bold, italic, and 22 pixels.



(a) (b) (c)

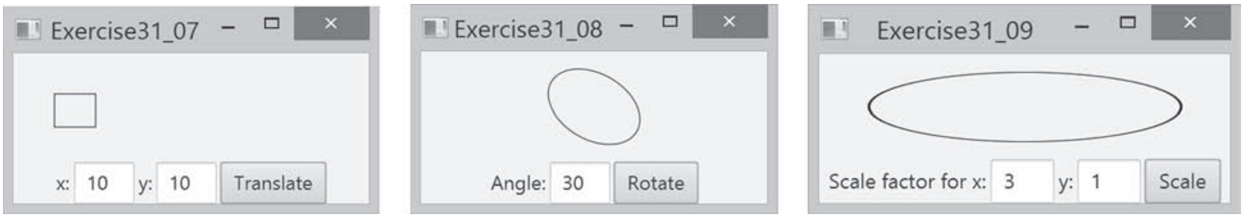
FIGURE 31.49 (a) Five texts are displayed with a random color and a specified font. (b) A path is displayed inside the circle. (c) Two circles are displayed in an oval.

***31.5** (*Cubic curve*) Write a program that creates two shapes: a circle and a path consisting of two cubic curves, as shown in Figure 31.49b.

***31.6** (*Eyes*) Write a program that displays two eyes in an oval, as shown in Figure 31.49c.

Sections 31.4

***31.7** (*Translation*) Write a program that displays a rectangle with upper-left corner point at (**40**, **40**), width **50**, and height **40**. Enter the values in the text fields *x* and *y* and press the *Translate* button to translate the rectangle to a new location, as shown in Figure 31.50a.



x: 10 y: 10

Angle: 30

Scale factor for x: 3 y: 1

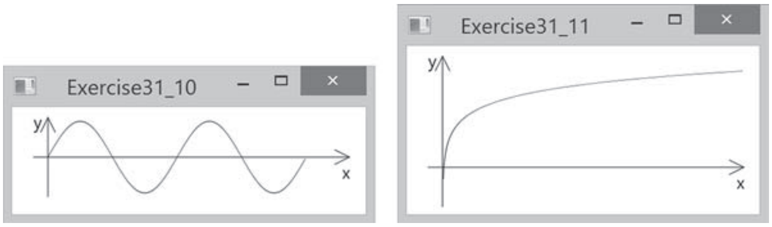
(a) (b) (c)

Figure 31.50 (a) Exercise 31.7 translates coordinates. (b) Exercise 31.8 rotates coordinates. (c) Exercise 31.9 scales coordinates.

***31.8** (*Rotation*) Write a program that displays an ellipse. The ellipse is centered in the pane with width **60** and height **40**. Enter the value in the text field Angle and press the *Rotate* button to rotate the ellipse, as shown in Figure 31.50b.

***31.9** (*Scale graphics*) Write a program that displays an ellipse. The ellipse is centered in the pane with width **60** and height **40**. Enter the scaling factors in the text fields and press the *Scale* button to scale the ellipse, as shown in Figure 31.50c.

***31.10** (*Plot the sine function*) Write a program that plots the sine function, as shown in Figure 31.51a.

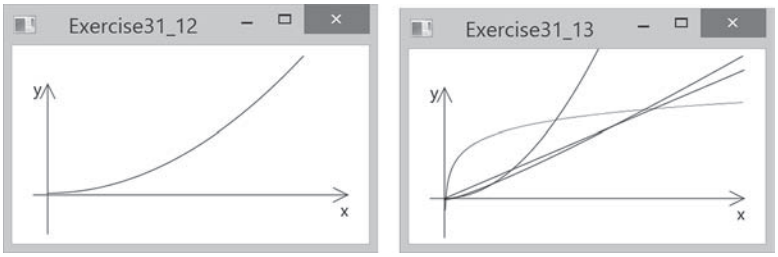


(a) (b)

Figure 31.51 (a) Exercise 31.10 displays a sine function. (b) Exercise 31.11 displays the log function.

***31.11** (*Plot the log function*) Write a program that plots the log function, as shown in Figure 31.51a.

***31.12** (*Plot the n^2 function*) Write a program that plots the n^2 function, as shown in Figure 31.51b 2a.


Exercise31_12
y
x
Exercise31_13
y
x

(a) (b)

FIGURE 31.52 (a) Exercise 31.13 displays the n^2 function. (b) Exercise 31.13 displays several functions.

***31.13** (*Plot the log, n, nlogn, and n^2 functions*) Write a program that plots the log, *n*, *n*log*n*, and n^2 functions, as shown in Figure 31.52b.

***31.14** (*Scale and rotate graphics*) Write a program that enables the user to scale and rotate the STOP sign, as shown in Figure 31.53. The user can press the UP/DOWN arrow key to increase/decrease the size and press the RIGHT/LEFT arrow key to rotate left or right.

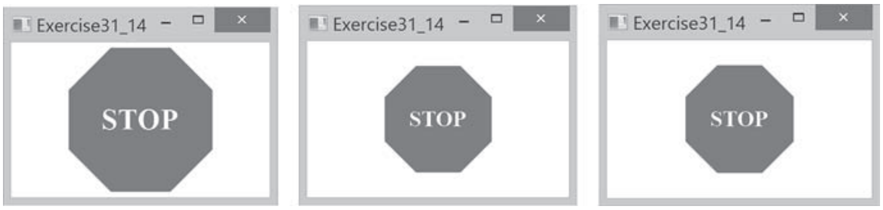
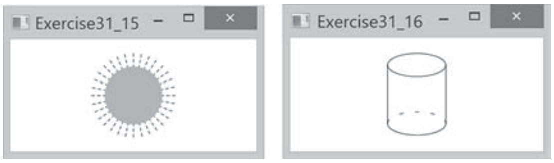

Exercise31_14
STOP
Exercise31_14
STOP
Exercise31_14
STOP

FIGURE 31.53 The program can rotate and scale the painting.

Sections 31.5

***31.15** (*Sunshine*) Write a program that displays a circle filled with a gradient color to animate a sun and display light rays coming out from the sun using dashed lines, as shown in Figure 31.54a.


Exercise31_15
Exercise31_16

(a) (b)

FIGURE 31.54 (a) Exercise 31.15 displays the sunshine. (b) Exercise 31.16 displays a cylinder.

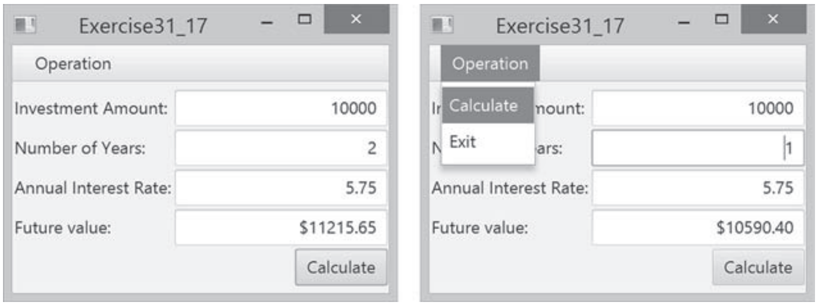
***31.16** (*Display a cylinder*) Write a program that displays a cylinder, as shown in Figure 31.54b. Use dashed strokes to draw the dashed arc.

Sections 31.6

***31.17** (*Create an investment value calculator*) Write a program that calculates the future value of an investment at a given interest rate for a specified number of years. The formula for the calculation is as follows:

$$\texttt{futureValue = investmentAmount} \times \texttt{(1 + monthlyInterestRate)}^{\texttt{years}\times\texttt{12}}$$

Use text fields for interest rate, investment amount, and years. Display the future amount in a text field when the user clicks the *Calculate* button or chooses Calculate from the Operation menu (see Figure 31.55). Click the Exit menu to exit the program.



Investment Amount: 10000
Number of Years: 2
Annual Interest Rate: 5.75
Future value: $11215.65

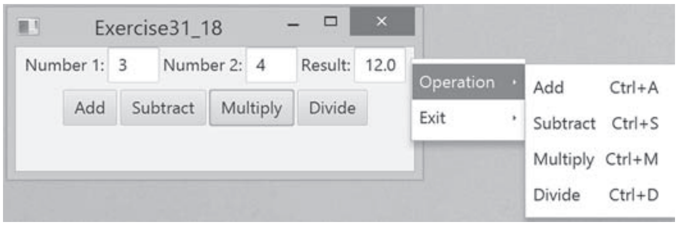
Investment Amount: 10000
Number of Years: 1
Annual Interest Rate: 5.75
Future value: $10590.40

Exercise31_17
Operation
Calculate
Exercise31_17
Operation
Calculate
Exit
Calculate

FIGURE 31.55 The user enters the investment amount, years, and interest rate to compute future value.

Sections 31.8

***31.18** (*Use popup menus*) Modify `Listing 31.9`, `MenuDemo.java`, to create a popup menu that contains the menus Operations and Exit, as shown in Figure 31.56. The popup is displayed when you click the right mouse button on the panel that contains the labels and the text fields.



Number 1: 3 Number 2: 4 Result: 12.0

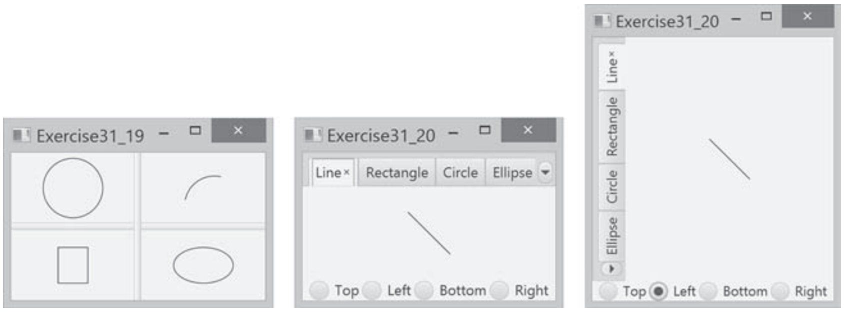
Exercise31_18
Add
Subtract
Multiply
Divide
Operation
Exit
Add Ctrl+A
Subtract Ctrl+S
Multiply Ctrl+M
Divide Ctrl+D

FIGURE 31.56 The popup menu contains the commands to perform arithmetic operations.

***31.19** (*Use* `SplitPane`) Create a program that displays four shapes in split panes, as shown in Figure 31.57a.

Sections 31.9

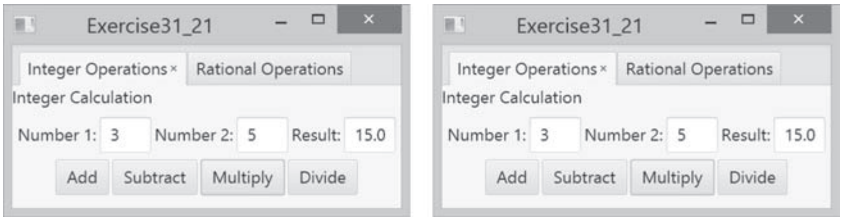
***31.20** (*Use tab panes*) Modify `Listing 31.12`, `TabPaneDemo.java`, to add a pane of radio buttons for specifying the tab placement of the tab pane, as shown in Figure 31.57b and c.


Exercise31_19
Exercise31_20
Line
Rectangle
Circle
Ellipse
○ Top ○ Left ○ Bottom ○ Right
Exercise31_20
Line
Rectangle
Circle
Ellipse
○ Top ● Left ○ Bottom ○ Right

(a) (b) (c)

FIGURE 31.57 (a) Four shapes are displayed in split panes. (b and c) The radio buttons let you choose the tab placement of the tabbed pane.

***31.21** (*Use tab panes*) Write a program using tab panes for performing integer and rational number arithmetic as shown in Figure 31.58.


Exercise31_21
Integer Operations
Rational Operations
Integer Calculation
Add
Subtract
Multiply
Divide
Exercise31_21
Integer Operations
Rational Operations
Integer Calculation
Add
Subtract
Multiply
Divide

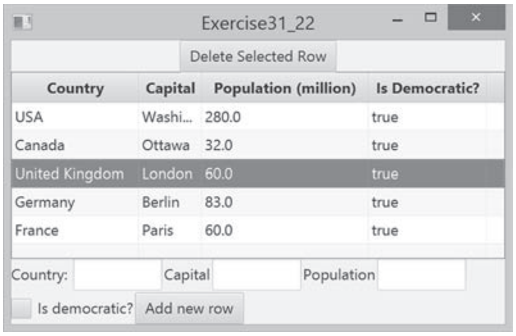
Number 1: 3 Number 2: 5 Result: 15.0

Number 1: 3 Number 2: 5 Result: 15.0

FIGURE 31.58 A tab pane is used to select panes that perform integer operations and rational number operations.

Sections 31.10

***31.22** (*Use table view*) Revise Listing 31.14 to add a button to delete the selected row from the table, as shown in Figure 31.59.


Exercise31_22
Delete Selected Row
Country:
Capital
Population
☐ Is democratic?
Add new row

Country	Capital	Population (million)	Is Democratic?
USA	Washi...	280.0	true
Canada	Ottawa	32.0	true
United Kingdom	London	60.0	true
Germany	Berlin	83.0	true
France	Paris	60.0	true

FIGURE 31.59 Clicking the *Delete Selected Row* button removes the selected row from the table.

CHAPTER 32

MULTITHREADING AND PARALLEL PROGRAMMING

Objectives

- To get an overview of multithreading (§32.2).
- To develop task classes by implementing the `Runnable` interface (§32.3).
- To create threads to run tasks using the `Thread` class (§32.3).
- To control threads using the methods in the `Thread` class (§32.4).
- To control animations using threads and use `Platform.runLater` to run the code in the application thread (§32.5).
- To execute tasks in a thread pool (§32.6).
- To use synchronized methods or blocks to synchronize threads to avoid race conditions (§32.7).
- To synchronize threads using locks (§32.8).
- To facilitate thread communications using conditions on locks (§§32.9 and 32.10).
- To use blocking queues (`ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue`) to synchronize access to a queue (§32.11).
- To restrict the number of concurrent accesses to a shared resource using semaphores (§32.12).
- To use the resource-ordering technique to avoid deadlocks (§32.13).
- To describe the life cycle of a thread (§32.14).
- To create synchronized collections using the static methods in the `Collections` class (§32.15).
- To develop parallel programs using the Fork/Join Framework (§32.16).

32.1 Introduction

Multithreading enables multiple tasks in a program to be executed concurrently.

multithreading

One of the powerful features of Java is its built-in support for *multithreading*—the concurrent running of multiple tasks within a program. In many programming languages, you have to invoke system-dependent procedures and functions to implement multithreading. This chapter introduces the concepts of threads and how multithreading programs can be developed in Java.

32.2 Thread Concepts

A program may consist of many tasks that can run concurrently. A thread is the flow of execution, from beginning to end, of a task.

thread
task

A *thread* provides the mechanism for running a task. With Java, you can launch multiple threads from a program concurrently. These threads can be executed simultaneously in multiprocessor systems, as shown in Figure 32.1a.

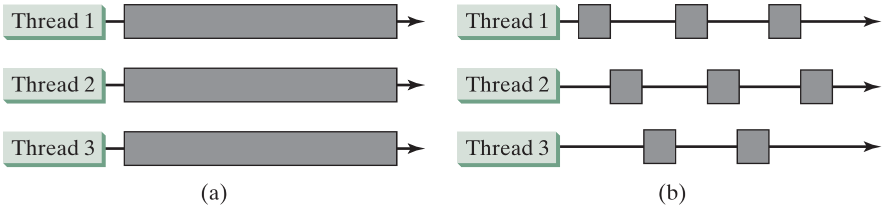


Figure 32.1 (a) Multiple threads running on multiple CPUs. (b) Multiple threads share a single CPU.

time sharing

In single-processor systems, as shown in Figure 32.1b, the multiple threads share CPU time, known as *time sharing*, and the operating system is responsible for scheduling and allocating resources to them. This arrangement is practical because most of the time the CPU is idle. It does nothing, for example, while waiting for the user to enter data.

Multithreading can make your program more responsive and interactive as well as enhance performance. For example, a good word processor lets you print or save a file while you are typing. In some cases, multithreaded programs run faster than single-threaded programs even on single-processor systems. Java provides exceptionally good support for creating and running threads, and for locking resources to prevent conflicts.

task
runnable object
thread

You can create additional threads to run concurrent tasks in the program. In Java, each task is an instance of the `Runnable` interface, also called a *runnable object*. A *thread* is essentially an object that facilitates the execution of a task.

32.2.1 Why is multithreading needed? How can multiple threads run simultaneously in a single-processor system?

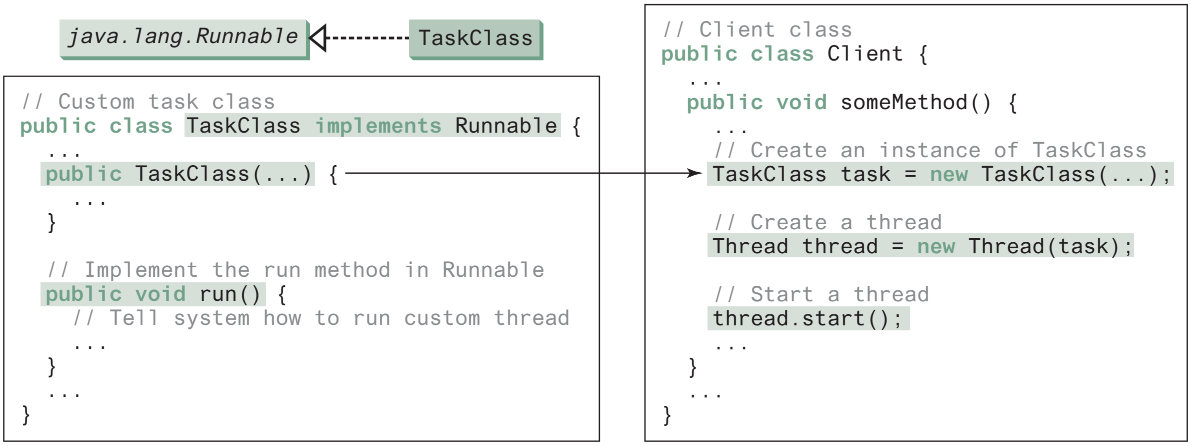
32.2.2 What is a runnable object? What is a thread?

32.3 Creating Tasks and Threads

A task class must implement the `Runnable` *interface. A task must be run from a thread.*

Runnable interface
run() method

Tasks are objects. To create tasks, you have to first define a class for tasks, which implements the `Runnable` interface. The `Runnable` interface is rather simple. All it contains is the `run()` method. You need to implement this method to tell the system how your thread is going to run. A template for developing a task class is shown in Figure 32.2a.



```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

(a) (b)

FIGURE 32.2 Define a task class by implementing the `Runnable` interface.

Once you have defined a `TaskClass`, you can create a task using its constructor. For example,

```
TaskClass task = new TaskClass(...);
```

Thread class

create a task

A task must be executed in a thread. The `Thread` class contains the constructors for creating threads and many useful methods for controlling threads. To create a thread for a task, use

```
Thread thread = new Thread(task);
```

create a thread

You can then invoke the `start()` method to tell the JVM that the thread is ready to run, as follows:

```
thread.start();
```

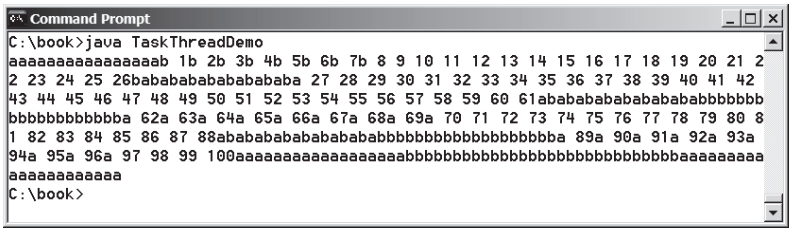
start a thread

The JVM will execute the task by invoking the task's `run()` method. Figure 32.2b outlines the major steps for creating a task, a thread, and starting the thread.

Listing 32.1 gives a program that creates three tasks and three threads to run them.

- The first task prints the letter *a* 100 times.
- The second task prints the letter *b* 100 times.
- The third task prints the integers 1 through 100.

When you run this program, the three threads will share the CPU and take turns printing letters and numbers on the console. Figure 32.3 shows a sample run of the program.



```
C:\book>java TaskThreadDemo
aaaaaaaaaaaaaaaab 1b 2b 3b 4b 5b 6b 7b 8 9 10 11 12 13 14 15 16 17 18 19 20 21 2
2 23 24 25 26bababababababababa 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61ababababababababbbbbbbb
bbbbbbbbbbbba 62a 63a 64a 65a 66a 67a 68a 69a 70 71 72 73 74 75 76 77 78 79 80 8
1 82 83 84 85 86 87 88ababababababababbbbbbbbbbbbbbbbbbbba 89a 90a 91a 92a 93a
94a 95a 96a 97 98 99 100aaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaa
aaaaaaaaaaaa
C:\book>
```

FIGURE 32.3 Tasks `printA`, `printB`, and `print100` are executed simultaneously to display the letter `a` 100 times, the letter `b` 100 times, and the numbers from 1 to 100.

LISTING 32.1 TaskThreadDemo.java

create tasks

create threads

start threads

task class

run

task class

run

```java
public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable printA = new PrintChar('a', 100);
    Runnable printB = new PrintChar('b', 100);
    Runnable print100 = new PrintNum(100);

    // Create threads
    Thread thread1 = new Thread(printA);
    Thread thread2 = new Thread(printB);
    Thread thread3 = new Thread(print100);

    // Start threads
    thread1.start();
    thread2.start();
    thread3.start();
  }
}

// The task for printing a character a specified number of times
class PrintChar implements Runnable {
  private char charToPrint; // The character to print
  private int times; // The number of times to repeat

  /** Construct a task with a specified character and number of
    * times to print the character
   */
  public PrintChar(char c, int t) {
    charToPrint = c;
    times = t;
  }

  @Override /** Override the run() method to tell the system
   * what task to perform
   */
  public void run() {
    for (int i = 0; i < times; i++) {
      System.out.print(charToPrint);
    }
  }
}

// The task class for printing numbers from 1 to n for a given n
class PrintNum implements Runnable {
  private int lastNum;

  /** Construct a task for printing 1, 2, ..., n */
  public PrintNum(int n) {
    lastNum = n;
  }

  @Override /** Tell the thread how to run */
  public void run() {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
    }
  }
}
```

The program creates three tasks (lines 4–6). To run them concurrently, three threads are created (lines 9–11). The `start()` method (lines 14–16) is invoked to start a thread that causes the `run()` method in the task to be executed. When the `run()` method completes, the thread terminates.

Because the first two tasks, `printA` and `printB`, have similar functionality, they can be defined in one task class `PrintChar` (lines 21–41). The `PrintChar` class implements `Runnable` and overrides the `run()` method (lines 36–40) with the print-character action. This class provides a framework for printing any single character a given number of times. The runnable objects, `printA` and `printB`, are instances of the `PrintChar` class.

The `PrintNum` class (lines 44–58) implements `Runnable` and overrides the `run()` method (lines 53–57) with the print-number action. This class provides a framework for printing numbers from 1 to *n*, for any integer *n*. The runnable object `print100` is an instance of the class `printNum` class.

Note

If you don't see the effect of these three threads running concurrently, increase the number of characters to be printed. For example, change line 4 to

effect of concurrency

```
Runnable printA = new PrintChar('a', 10000);
```

Important Note

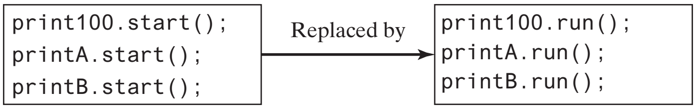
The `run()` method in a task specifies how to perform the task. This method is automatically invoked by the JVM. You should not invoke it. Invoking `run()` directly merely executes this method in the same thread; no new thread is started.

`run()` method

Check Point

32.3.1 How do you define a task class? How do you create a thread for a task?

32.3.2 What would happen if you replace the `start()` method with the `run()` method in lines 14–16 in Listing 32.1?



```
print100.start();
printA.start();
printB.start();
```

Replaced by

```
print100.run();
printA.run();
printB.run();
```

32.3.3 What is wrong in the following two programs? Correct the errors.

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() {
    Test task = new Test();
    new Thread(task).start();
  }

  public void run() {
    System.out.println("test");
  }
}
```

(a)

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() {
    Thread t = new Thread(this);
    t.start();
    t.start();
  }

  public void run() {
    System.out.println("test");
  }
}
```

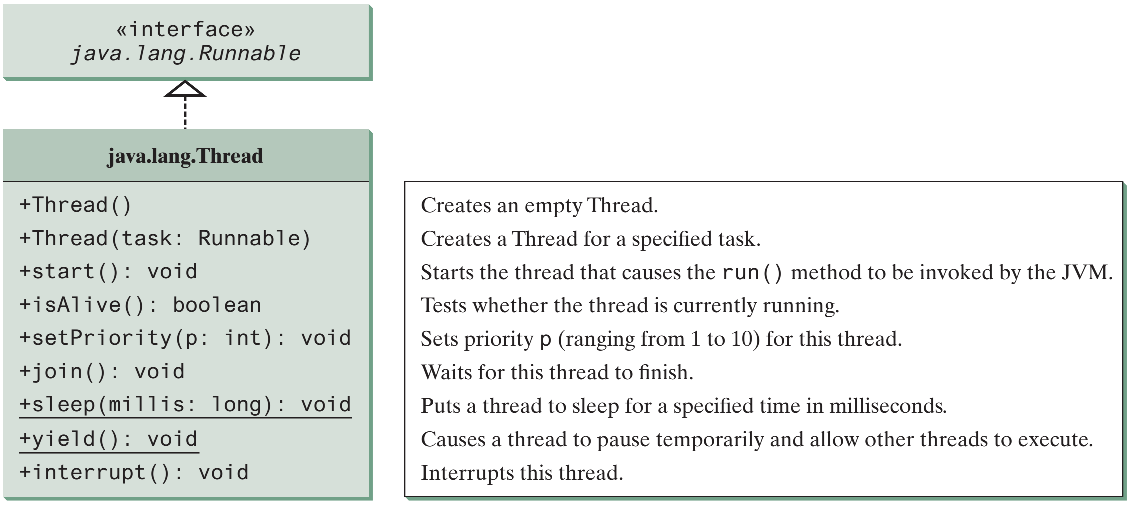
(b)

32.4 The Thread Class

Key Point

The **Thread** *class contains the constructors for creating threads for tasks and the methods for controlling threads.*

Figure 32.4 shows the class diagram for the **Thread** class.



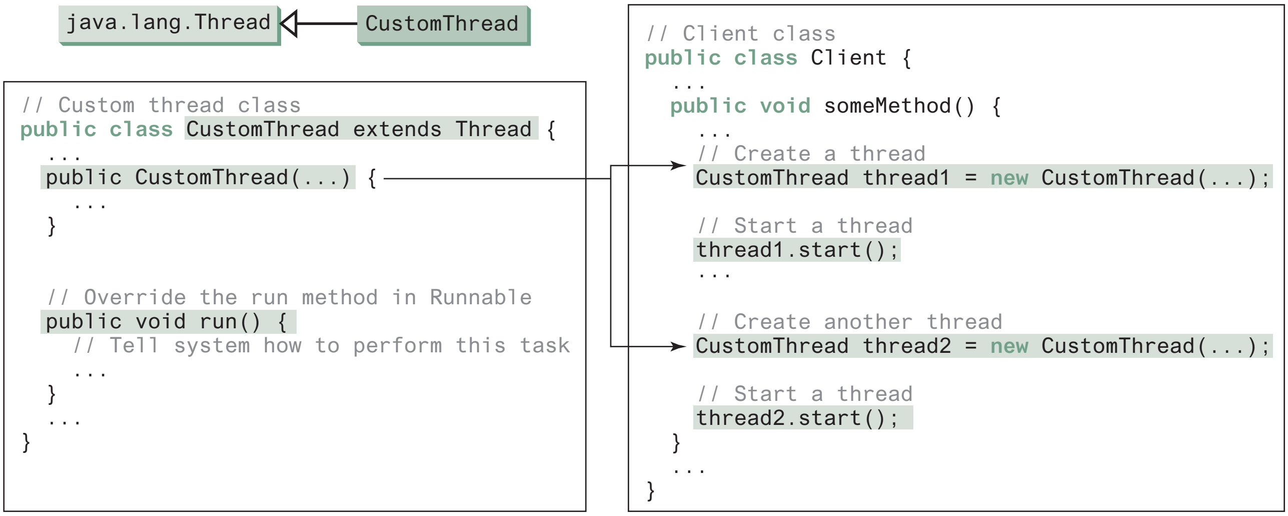
java.lang.Thread	
`+Thread()`	Creates an empty Thread.
`+Thread(task: Runnable)`	Creates a Thread for a specified task.
`+start(): void`	Starts the thread that causes the `run()` method to be invoked by the JVM.
`+isAlive(): boolean`	Tests whether the thread is currently running.
`+setPriority(p: int): void`	Sets priority `p` (ranging from 1 to 10) for this thread.
`+join(): void`	Waits for this thread to finish.
`+sleep(millis: long): void`	Puts a thread to sleep for a specified time in milliseconds.
`+yield(): void`	Causes a thread to pause temporarily and allow other threads to execute.
`+interrupt(): void`	Interrupts this thread.

FIGURE 32.4 The **Thread** class contains the methods for controlling threads.

Note

Since the **Thread** class implements **Runnable**, you could define a class that extends **Thread** and implements the **run** method, as shown in Figure 32.5a, then create an object from the class and invoke its **start** method in a client program to start the thread, as shown in Figure 32.5b.

separating task from thread



```
// Custom thread class
public class CustomThread extends Thread {
  ...
  public CustomThread(...) {
    ...
  }

  // Override the run method in Runnable
  public void run() {
    // Tell system how to perform this task
    ...
  }
  ...
}
```

(a)

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create a thread
    CustomThread thread1 = new CustomThread(...);

    // Start a thread
    thread1.start();
    ...

    // Create another thread
    CustomThread thread2 = new CustomThread(...);

    // Start a thread
    thread2.start();
  }
  ...
}
```

(b)

FIGURE 32.5 Define a thread class by extending the **Thread** class.

This approach is, however, not recommended because it mixes the task and the mechanism of running the task. Separating the task from the thread is a preferred design.

Note

The `Thread` class also contains the `stop()`, `suspend()`, and `resume()` methods. As of Java 2, these methods were *deprecated* (or *outdated*) because they are known to be inherently unsafe. Instead of using the `stop()` method, you should assign `null` to a `Thread` variable to indicate that it has stopped.

deprecated method

You can use the `yield()` method to temporarily release time for other threads. For example, suppose that you modify the code in the `run()` method in lines 53–57 for `PrintNum` in Listing 32.1 as follows:

`yield()`

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    Thread.yield();
  }
}
```

Every time a number is printed, the thread of the `print100` task is yielded to other threads.

The `sleep(long millis)` method puts the thread to sleep for a specified time in milliseconds to allow other threads to execute. For example, suppose that you modify the code in lines 53–57 in Listing 32.1 as follows:

`sleep(long)`

```
public void run() {
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i >= 50) Thread.sleep(1);
    }
  }
  catch (InterruptedException ex) {
  }
}
```

Every time a number (`>= 50`) is printed, the thread of the `print100` task is put to sleep for 1 millisecond.

The `sleep` method may throw an `InterruptedException`, which is a checked exception. Such an exception may occur when a sleeping thread's `interrupt()` method is called. The `interrupt()` method is very rarely invoked on a thread, so an `InterruptedException` is unlikely to occur. But since Java forces you to catch checked exceptions, you have to put it in a `try-catch` block. If a `sleep` method is invoked in a loop, you should wrap the loop in a `try-catch` block, as shown in (a) below. If the loop is outside the `try-catch` block, as shown in (b), the thread may continue to execute even though it is being interrupted.

`InterruptedException`

```
public void run() {
  try {
    while (...) {
      ...
      Thread.sleep(1000);
    }
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

(a) Correct

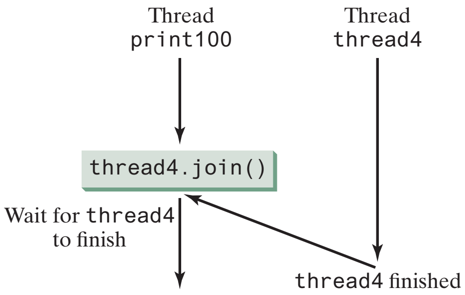
```
public void run() {
  while (...) {
    try {
      ...
      Thread.sleep(sleepTime);
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}
```

(b) Incorrect

`join()`

You can use the `join()` method to force one thread to wait for another thread to finish. For example, suppose that you modify the code in lines 53–57 in Listing 32.1 as follows:

```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print (" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
}
```



A new `thread4` is created and it prints character *c* 40 times. The numbers from `50` to `100` are printed after thread `thread4` is finished.

`setPriority(int)`

Java assigns every thread a priority. By default, a thread inherits the priority of the thread that spawned it. You can increase or decrease the priority of any thread by using the `setPriority` method and you can get the thread's priority by using the `getPriority` method. Priorities are numbers ranging from `1` to `10`. The `Thread` class has the `int` constants `MIN_PRIORITY`, `NORM_PRIORITY`, and `MAX_PRIORITY`, representing `1`, `5`, and `10`, respectively. The priority of the main thread is `Thread.NORM_PRIORITY`.

round-robin scheduling

The JVM always picks the currently runnable thread with the highest priority. A lower priority thread can run only when no higher priority threads are running. If all runnable threads have equal priorities, each is assigned an equal portion of the CPU time in a circular queue. This is called *round-robin scheduling*. For example, suppose that you insert the following code in line 16 in Listing 32.1:

```
thread3.setPriority(Thread.MAX_PRIORITY);
```

The thread for the `print100` task will be finished first.

Tip

The priority numbers may be changed in a future version of Java. To minimize the impact of any changes, use the constants in the `Thread` class to specify thread priorities.

contention or starvation

Tip

A thread may never get a chance to run if there is always a higher priority thread running or a same-priority thread that never yields. This situation is known as *contention* or *starvation*. To avoid contention, the thread with higher priority must periodically invoke the `sleep` or `yield` method to give a thread with a lower or the same priority a chance to run.

Check Point

32.4.1 Which of the following methods are instance methods in `java.lang.Thread`? Which method may throw an `InterruptedException`? Which of them are deprecated in Java?

`run`, `start`, `stop`, `suspend`, `resume`, `sleep`, `interrupt`, `yield`, `join`

32.4.2 If a loop contains a method that throws an `InterruptedException`, why should the loop be placed inside a `try-catch` block?

32.4.3 How do you set a priority for a thread? What is the default priority?

32.5 Animation Using Threads and the `Platform.runLater` Method

Key Point

You can use a thread to control an animation and run the code in JavaFX GUI thread using the `Platform.runLater` *method.*

The use of a `Timeline` object to control animations was introduced in Section 15.11, Animation. Alternatively, you can also use a thread to control animation. Listing 32.2 gives an example that displays flashing text on a label, as shown in Figure 32.6.

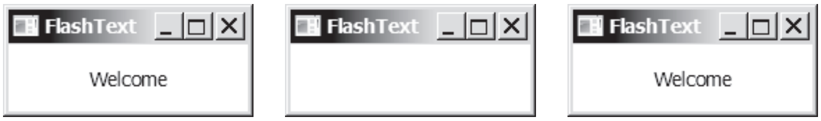


Figure 32.6 The text "Welcome" blinks.

Listing 32.2 `FlashText.java`

```java
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class FlashText extends Application {
  private String text = "";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    StackPane pane = new StackPane();
    Label lblText = new Label("Programming is fun");
    pane.getChildren().add(lblText);

    new Thread(new Runnable() {
      @Override
      public void run() {
        try {
          while (true) {
            if (lblText.getText().trim().length() == 0)
              text = "Welcome";
            else
              text = "";

            Platform.runLater(new Runnable() { // Run from JavaFX GUI
              @Override
              public void run() {
                lblText.setText(text);
              }
            });

            Thread.sleep(200);
          }
        }
        catch (InterruptedException ex) {
        }
      }
    }).start();

```

create a label
label in a pane
create a thread
run thread
change text
`Platform.runLater`
update GUI
sleep

```
42      // Create a scene and place it in the stage
43      Scene scene = new Scene(pane, 200, 50);
44      primaryStage.setTitle("FlashText"); // Set the stage title
45      primaryStage.setScene(scene); // Place the scene in the stage
46      primaryStage.show(); // Display the stage
47    }
48  }
```

The program creates a `Runnable` object in an anonymous inner class (lines 17–40). This object is started in line 40 and runs continuously to change the text in the label. It sets a text in the label if the label is blank (line 23) and sets its text blank (line 25) if the label has a text. The text is set and unset to simulate a flashing effect.

JavaFX application thread

JavaFX GUI is run from the *JavaFX application thread*. The flashing control is run from a separate thread. The code in a nonapplication thread cannot update GUI in the application thread. To update the text in the label, a new `Runnable` object is created in lines 27–32. Invoking `Platform.runLater(Runnable r)` tells the system to run this `Runnable` object in the application thread.

`Platform.runLater`

The anonymous inner classes in this program can be simplifed using lambda expressions as follows:

```
new Thread(() -> { // lambda expression
  try {
    while (true) {
      if (lblText.getText().trim().length() == 0)
        text = "Welcome";
      else
        text = "";

      Platform.runLater(() -> lblText.setText(text)); // lambda exp

      Thread.sleep(200);
    }
  }
  catch (InterruptedException ex) {
  }
}).start();
```

Check Point

32.5.1 What causes the text to flash?

32.5.2 Is an instance of `FlashText` a runnable object?

32.5.3 What is the purpose of using `Platform.runLater`?

32.5.4 Can you replace the code in lines 27–32 using the following code?

```
Platform.runLater(e -> lblText.setText(text));
```

32.5.5 What happens if line 34 (`Thread.sleep(200)`) is not used?

32.5.6 There is an issue in Listing 16.9, ListViewDemo. If you press the CTRL key and select Canada, Demark, and China in this order, you will get an `ArrayIndexOutBoundsException`. What is the reason for this error and how do you fix it? (Thanks to Henri Heimonen of Finland for contributing this question).

32.6 Thread Pools

Key Point

A thread pool can be used to execute tasks efficiently.

In Section 32.3, Creating Tasks and Threads, you learned how to define a task class by implementing `java.lang.Runnable`, and how to create a thread to run a task like this:

```
Runnable task = new TaskClass(...);
new Thread(task).start();
```

This approach is convenient for a single task execution, but it is not efficient for a large number of tasks because you have to create a thread for each task. Starting a new thread for each task could limit throughput and cause poor performance. Using a *thread pool* is an ideal way to manage the number of tasks executing concurrently. Java provides the `Executor` interface for executing tasks in a thread pool and the `ExecutorService` interface for managing and controlling tasks. `ExecutorService` is a subinterface of `Executor`, as shown in Figure 32.7.

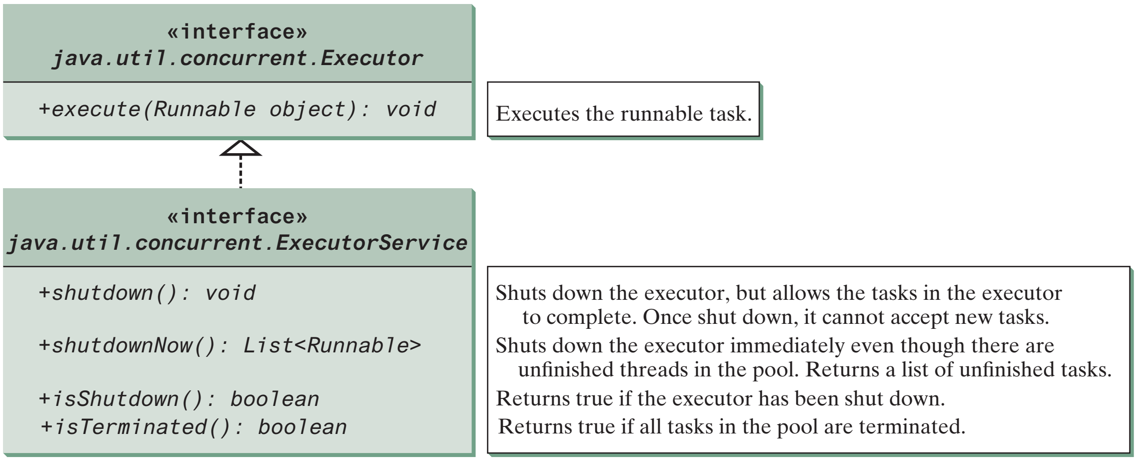

«interface»
java.util.concurrent.Executor
+execute(Runnable object): void
Executes the runnable task.
«interface»
java.util.concurrent.ExecutorService
+shutdown(): void
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
+isTerminated(): boolean
Shuts down the executor, but allows the tasks in the executor to complete. Once shut down, it cannot accept new tasks.
Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.
Returns true if the executor has been shut down.
Returns true if all tasks in the pool are terminated.

FIGURE 32.7 The `Executor` interface executes threads and the `ExecutorService` subinterface manages threads.

To create an `Executor` object, use the static methods in the `Executors` class, as shown in Figure 32.8. The `newFixedThreadPool(int)` method creates a fixed number of threads in a pool. If a thread completes executing a task, it can be reused to execute another task. If a thread terminates due to a failure prior to shutdown, a new thread will be created to replace it if all the threads in the pool are not idle and there are tasks waiting for execution. The `newCachedThreadPool()` method creates a new thread if all the threads in the pool are not idle and there are tasks waiting for execution. A thread in a cached pool will be terminated if it has not been used for 60 seconds. A cached pool is efficient for many short tasks.

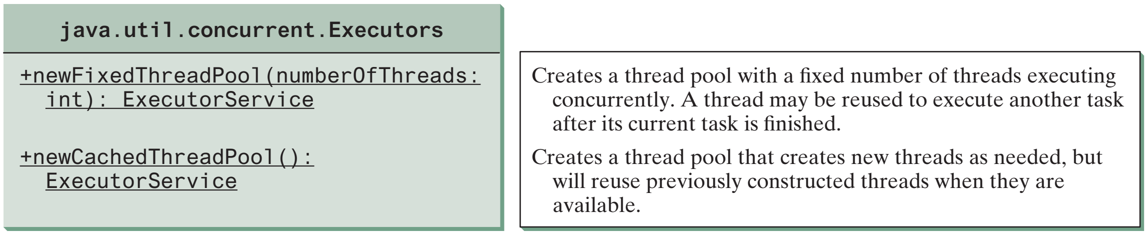

java.util.concurrent.Executors
+newFixedThreadPool(numberOfThreads: int): ExecutorService
+newCachedThreadPool(): ExecutorService
Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.
Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

FIGURE 32.8 The `Executors` class provides static methods for creating `Executor` objects.

Listing 32.3 shows how to rewrite Listing 32.1 using a thread pool.

LISTING 32.3 ExecutorDemo.java

```
import java.util.concurrent.*;

public class ExecutorDemo {
```

create executor

submit task

shut down executor

```
 4    public static void main(String[] args) {
 5      // Create a fixed thread pool with maximum three threads
 6      ExecutorService executor = Executors.newFixedThreadPool(3);
 7
 8      // Submit runnable tasks to the executor
 9      executor.execute(new PrintChar('a', 100));
10      executor.execute(new PrintChar('b', 100));
11      executor.execute(new PrintNum(100));
12
13      // Shut down the executor
14      executor.shutdown();
15    }
16  }
```

Line 6 creates a thread pool executor with a total of three threads maximum. Classes `PrintChar` and `PrintNum` are defined in Listing 32.1. Line 9 creates a task, `new PrintChar('a', 100)`, and adds it to the pool. Similarly, another two runnable tasks are created and added to the same pool in lines 10 and 11. The executor creates three threads to execute three tasks concurrently.

Suppose you replace line 6 with

```
ExecutorService executor = Executors.newFixedThreadPool(1);
```

What will happen? The three runnable tasks will be executed sequentially because there is only one thread in the pool.

Suppose you replace line 6 with

```
ExecutorService executor = Executors.newCachedThreadPool();
```

What will happen? New threads will be created for each waiting task, so all the tasks will be executed concurrently.

The `shutdown()` method in line 14 tells the executor to shut down. No new tasks can be accepted, but any existing tasks will continue to finish.

Tip

If you need to create a thread for just one task, use the `Thread` class. If you need to create threads for multiple tasks, it is better to use a thread pool.

Check Point

32.6.1 What are the benefits of using a thread pool?

32.6.2 How do you create a thread pool with three fixed threads? How do you submit a task to a thread pool? How do you know that all the tasks are finished?

Key Point

32.7 Thread Synchronization

Thread synchronization is to coordinate the execution of the dependent threads.

A shared resource may become corrupted if it is accessed simultaneously by multiple threads. The following example demonstrates the problem.

Suppose that you create and launch 100 threads, each of which adds a penny to an account. Define a class named `Account` to model the account, a class named `AddAPennyTask` to add a penny to the account, and a main class that creates and launches threads. The relationships of these classes are shown in Figure 32.9. The program is given in Listing 32.4.

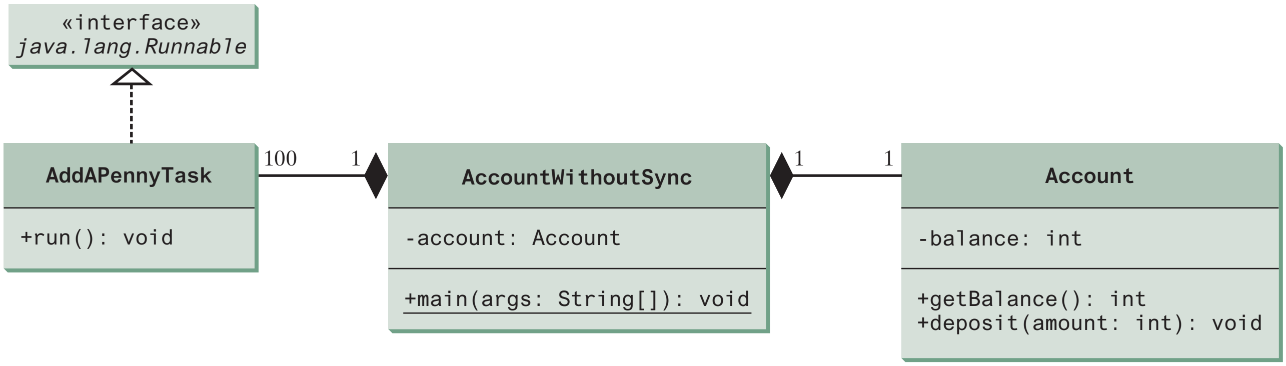

«interface»
java.lang.Runnable
AddAPennyTask
+run(): void
100
1
AccountWithoutSync
-account: Account
+main(args: String[]): void
1
1
Account
-balance: int
+getBalance(): int
+deposit(amount: int): void

FIGURE 32.9 `AccountWithoutSync` contains an instance of `Account` and 100 threads of `AddAPennyTask`.

LISTING 32.4 AccountWithoutSync.java

```
import java.util.concurrent.*;

public class AccountWithoutSync {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  private static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }

  // An inner class for account
  private static class Account {
    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void deposit(int amount) {
      int newBalance = balance + amount;

      // This delay is deliberately added to magnify the
```

create executor

submit task

shut down executor

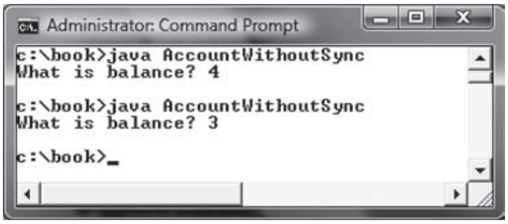
wait for all tasks to terminate

```
42         // data-corruption problem and make it easy to see.
43         try {
44           Thread.sleep(5);
45         }
46         catch (InterruptedException ex) {
47         }
48
49         balance = newBalance;
50       }
51     }
52   }
```

The classes **AddAPennyTask** and **Account** in lines 24–51 are inner classes. Line 4 creates an **Account** with initial balance **0**. Line 11 creates a task to add a penny to the account and submits the task to the executor. Line 11 is repeated 100 times in lines 10–12. The program repeatedly checks whether all tasks are completed in lines 17 and 18. The account balance is displayed in line 20 after all tasks are completed.

The program creates 100 threads executed in a thread pool **executor** (lines 10–12). The **isTerminated()** method (line 17) is used to test whether all the threads in the pool are terminated.

The balance of the account is initially **0** (line 32). When all the threads are finished, the balance should be **100** but the output is unpredictable. As can be seen in Figure 32.10, the answers are wrong in the sample run. This demonstrates the data-corruption problem that occurs when all the threads have access to the same data source simultaneously.



```
c:\book>java AccountWithoutSync
What is balance? 4

c:\book>java AccountWithoutSync
What is balance? 3

c:\book>_
```

FIGURE 32.10 The **AccountWithoutSync** program causes data inconsistency.

Lines 39–49 could be replaced by one statement:

```
balance = balance + amount;
```

It is highly unlikely, although plausible, that the problem can be replicated using this single statement. The statements in lines 39–49 are deliberately designed to magnify the data-corruption problem and make it easy to see. If you run the program several times but still do not see the problem, increase the sleep time in line 44. This will increase the chances for showing the problem of data inconsistency.

What, then, caused the error in this program? A possible scenario is shown in Figure 32.11.

Step	Balance	Task 1	Task 2
1	0	`newBalance = balance + 1;`	
2	0		`newBalance = balance + 1;`
3	1	`balance = newBalance;`	
4	1		`balance = newBalance;`

FIGURE 32.11 Task 1 and Task 2 both add 1 to the same balance.

In Step 1, Task 1 gets the balance from the account. In Step 2, Task 2 gets the same balance from the account. In Step 3, Task 1 writes a new balance to the account. In Step 4, Task 2 writes a new balance to the account.

The effect of this scenario is that Task 1 does nothing because in Step 4, Task 2 overrides Task 1's result. Obviously, the problem is that Task 1 and Task 2 are accessing a common resource in a way that causes a conflict. This is a common problem, known as a *race condition*, in multithreaded programs. A class is said to be *thread-safe* if an object of the class does not cause a race condition in the presence of multiple threads. As demonstrated in the preceding example, the `Account` class is not thread-safe.

race condition

thread-safe

32.7.1 The `synchronized` Keyword

To avoid race conditions, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program, known as the *critical region*. The critical region in Listing 32.4 is the entire `deposit` method. You can use the keyword `synchronized` to synchronize the method so that only one thread can access the method at a time. There are several ways to correct the problem in Listing 32.4. One approach is to make `Account` thread-safe by adding the keyword `synchronized` in the `deposit` method in line 38, as follows:

critical region

```
public synchronized void deposit(double amount)
```

A synchronized method acquires a lock before it executes. A lock is a mechanism for exclusive use of a resource. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static method, the lock is on the class. If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released. Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

With the `deposit` method synchronized, the preceding scenario cannot happen. If Task 1 enters the method, Task 2 is blocked until Task 1 finishes the method, as shown in Figure 32.12.

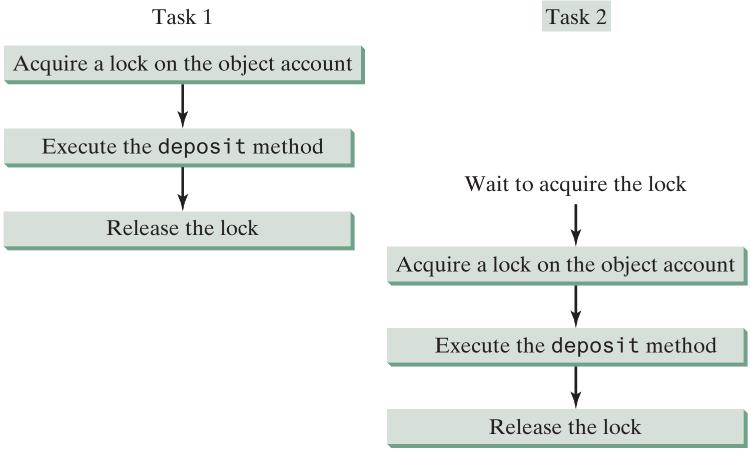

Task 1
Acquire a lock on the object account
Execute the deposit method
Release the lock
Task 2
Wait to acquire the lock
Acquire a lock on the object account
Execute the deposit method
Release the lock

FIGURE 32.12 Task 1 and Task 2 are synchronized.

32.7.2 Synchronizing Statements

Invoking a synchronized instance method of an object acquires a lock on the object, and invoking a synchronized static method of a class acquires a lock on the class. A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block synchronized block

of the code in a method. This block is referred to as a *synchronized block*. The general form of a synchronized statement is as follows:

```
synchronized (expr) {
  statements;
}
```

The expression `expr` must evaluate to an object reference. If the object is already locked by another thread, the thread is blocked until the lock is released. When a lock is obtained on the object, the statements in the synchronized block are executed and then the lock is released.

Synchronized statements enable you to synchronize part of the code in a method instead of the entire method. This increases concurrency. You can make Listing 32.4 thread-safe by placing the statement in line 26 inside a synchronized block:

```
synchronized (account) {
  account.deposit(1);
}
```

Note

Any synchronized instance method can be converted into a synchronized statement. For example, the following synchronized instance method in (a) is equivalent to (b):

```
public synchronized void xMethod() {
  // method body
}
```

(a)

```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

(b)

Check Point

32.7.1 Give some examples of possible resource corruption when running multiple threads. How do you synchronize conflicting threads?

32.7.2 Suppose you place the statement in line 26 of Listing 32.4 inside a synchronized block to avoid race conditions, as follows:

```
synchronized (this) {
  account.deposit(1);
}
```

Will it work?

32.8 Synchronization Using Locks

Key Point

Locks and conditions can be explicitly used to synchronize threads.

Recall that in Listing 32.4, 100 tasks deposit a penny to the same account concurrently, which causes conflicts. To avoid it, you use the `synchronized` keyword in the `deposit` method, as follows:

```
public synchronized void deposit(double amount)
```

lock

A synchronized instance method implicitly acquires a *lock* on the instance before it executes the method.

Java enables you to acquire locks explicitly, which give you more control for coordinating threads. A lock is an instance of the `Lock` interface, which defines the methods for acquiring and releasing locks, as shown in Figure 32.13. A lock may also use the `newCondition()` method to create any number of `Condition` objects, which can be used for thread communications.

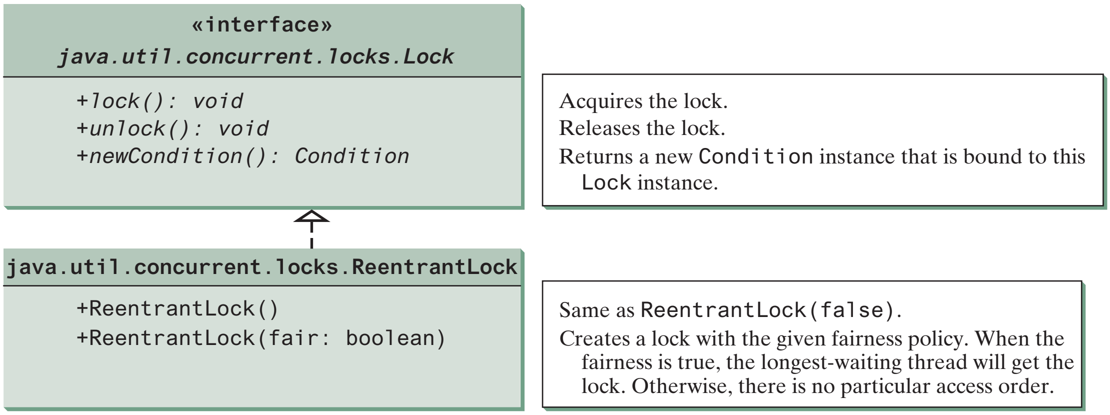

«interface»
java.util.concurrent.locks.Lock
+lock(): void
+unlock(): void
+newCondition(): Condition
Acquires the lock.
Releases the lock.
Returns a new Condition instance that is bound to this Lock instance.
java.util.concurrent.locks.ReentrantLock
+ReentrantLock()
+ReentrantLock(fair: boolean)
Same as ReentrantLock(false).
Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

FIGURE 32.13 The `ReentrantLock` class implements the `Lock` interface to represent a lock.

`ReentrantLock` is a concrete implementation of `Lock` for creating mutually exclusive locks. You can create a lock with the specified *fairness policy*. True fairness policies guarantee that the longest waiting thread will obtain the lock first. False fairness policies grant a lock to a waiting thread arbitrarily. Programs using fair locks accessed by many threads may have poorer overall performance than those using the default setting, but they have smaller variances in times to obtain locks and prevent starvation.

fairness policy

Listing 32.5 revises the program in Listing 32.7 to synchronize the account modification using explicit locks.

LISTING 32.5 AccountWithSyncUsingLock.java

package for locks

```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AccountWithSyncUsingLock {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  public static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }

```

create a lock

acquire the lock

release the lock

```
31   // An inner class for Account
32   public static class Account {
33     private static Lock lock = new ReentrantLock(); // Create a lock
34     private int balance = 0;
35
36     public int getBalance() {
37       return balance;
38     }
39
40     public void deposit(int amount) {
41       lock.lock(); // Acquire the lock
42
43       try {
44         int newBalance = balance + amount;
45
46         // This delay is deliberately added to magnify the
47         // data-corruption problem and make it easy to see.
48         Thread.sleep(5);
49
50         balance = newBalance;
51       }
52       catch (InterruptedException ex) {
53       }
54       finally {
55         lock.unlock(); // Release the lock
56       }
57     }
58   }
59 }
```

Line 33 creates a lock, line 41 acquires the lock, and line 55 releases the lock.

Tip

It is a good practice to always immediately follow a call to `lock()` with a `try-catch` block and release the lock in the `finally` clause, as shown in lines 41–56, to ensure that the lock is always released.

Listing 32.5 can be implemented using a synchronize method for `deposit` rather than using a lock. In general, using `synchronized` methods or statements is simpler than using explicit locks for mutual exclusion. However, using explicit locks is more intuitive and flexible to synchronize threads with conditions, as you will see in the next section.

Check Point

32.8.1 How do you create a lock object? How do you acquire a lock and release a lock?

32.9 Cooperation among Threads

Key Point

Conditions on locks can be used to coordinate thread interactions.

condition

Thread synchronization suffices to avoid race conditions by ensuring the mutual exclusion of multiple threads in the critical region, but sometimes you also need a way for threads to cooperate. *Conditions* can be used to facilitate communications among threads. A thread can specify what to do under a certain condition. Conditions are objects created by invoking the `newCondition()` method on a `Lock` object. Once a condition is created, you can use its `await()`, `signal()`, and `signalAll()` methods for thread communications, as shown in Figure 32.14. The `await()` method causes the current thread to wait until the condition is signaled. The `signal()` method wakes up one waiting thread, and the `signalAll()` method wakes all waiting threads.

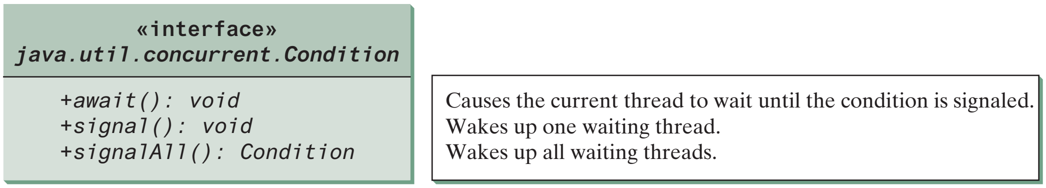

«interface»
java.util.concurrent.Condition
+await(): void
+signal(): void
+signalAll(): Condition
Causes the current thread to wait until the condition is signaled.
Wakes up one waiting thread.
Wakes up all waiting threads.

FIGURE 32.14 The `Condition` interface defines the methods for performing synchronization.

thread cooperation example

Let us use an example to demonstrate thread communications. Suppose you create and launch two tasks: one that deposits into an account, and one that withdraws from the same account. The withdraw task has to wait if the amount to be withdrawn is more than the current balance. Whenever new funds are deposited into the account, the deposit task notifies the withdraw thread to resume. If the amount is still not enough for a withdrawal, the withdraw thread has to continue to wait for a new deposit.

To synchronize the operations, use a lock with a condition: `newDeposit` (i.e., new deposit added to the account). If the balance is less than the amount to be withdrawn, the withdraw task will wait for the `newDeposit` condition. When the deposit task adds money to the account, the task signals the waiting withdraw task to try again. The interaction between the two tasks is shown in Figure 32.15.

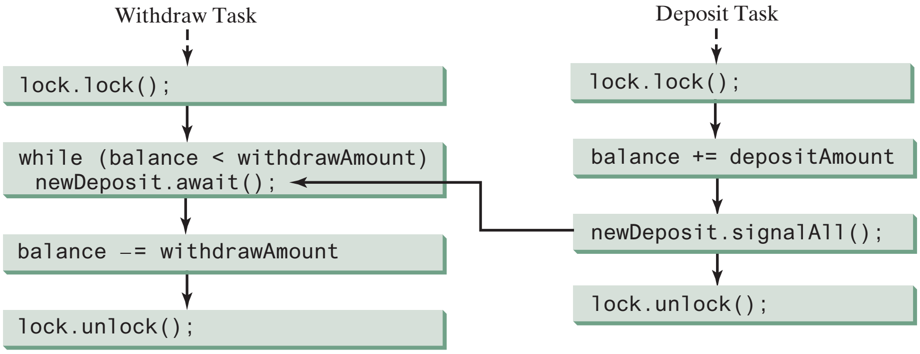
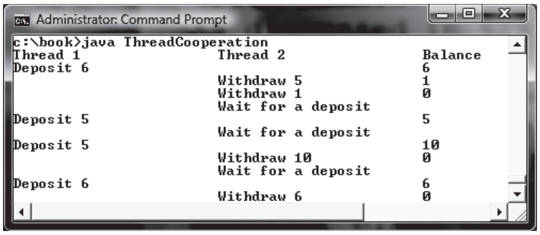

Withdraw Task
lock.lock();
while (balance < withdrawAmount)
newDeposit.await();
balance -= withdrawAmount
lock.unlock();
Deposit Task
lock.lock();
balance += depositAmount
newDeposit.signalAll();
lock.unlock();

FIGURE 32.15 The condition `newDeposit` is used for communications between the two threads.

You create a condition from a `Lock` object. To use a condition, you have to first obtain a lock. The `await()` method causes the thread to wait and automatically releases the lock on the condition. Once the condition is right, the thread reacquires the lock and continues executing.

Assume the initial balance is `0` and the amount to deposit and withdraw are randomly generated. Listing 32.6 gives the program. A sample run of the program is shown in Figure 32.16.


Administrator: Command Prompt

```
c:\book>java ThreadCooperation
Thread 1                Thread 2                Balance
Deposit 6                                       6
                        Withdraw 5              1
                        Withdraw 1              0
                        Wait for a deposit
Deposit 5                                       5
                        Wait for a deposit
Deposit 5                                       10
                        Withdraw 10             0
                        Wait for a deposit
Deposit 6                                       6
                        Withdraw 6              0
```

FIGURE 32.16 The withdraw task waits if there are not sufficient funds to withdraw.

LISTING 32.6 ThreadCooperation.java

```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ThreadCooperation {
  private static Account account = new Account();

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new DepositTask());
    executor.execute(new WithdrawTask());
    executor.shutdown();

    System.out.println("Thread 1\t\tThread 2\t\tBalance");
  }

  public static class DepositTask implements Runnable {
    @Override // Keep adding an amount to the account
    public void run() {
      try { // Purposely delay it to let the withdraw method proceed
        while (true) {
          account.deposit((int)(Math.random() * 10) + 1);
          Thread.sleep(1000);
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

  public static class WithdrawTask implements Runnable {
    @Override // Keep subtracting an amount from the account
    public void run() {
      while (true) {
        account.withdraw((int)(Math.random() * 10) + 1);
      }
    }
  }

  // An inner class for account
  private static class Account {
    // Create a new lock
    private static Lock lock = new ReentrantLock();

    // Create a condition
    private static Condition newDeposit = lock.newCondition();

    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void withdraw(int amount) {
      lock.lock(); // Acquire the lock
      try {
        while (balance < amount) {
```

create two threads

create a lock

create a condition

acquire the lock

```
59            System.out.println("\t\t\tWait for a deposit");
60            newDeposit.await();
61          }
62
63          balance -= amount;
64          System.out.println("\t\t\tWithdraw " + amount +
65            "\t\t" + getBalance());
66        }
67        catch (InterruptedException ex) {
68          ex.printStackTrace();
69        }
70        finally {
71          lock.unlock(); // Release the lock
72        }
73      }
74
75      public void deposit(int amount) {
76        lock.lock(); // Acquire the lock
77        try {
78          balance += amount;
79          System.out.println("Deposit " + amount +
80            "\t\t\t\t\t" + getBalance());
81
82          // Signal thread waiting on the condition
83          newDeposit.signalAll();
84        }
85        finally {
86          lock.unlock(); // Release the lock
87        }
88      }
89    }
90  }
```

wait on the condition

release the lock

acquire the lock

signal threads

release the lock

The example creates a new inner class named `Account` to model the account with two methods, `deposit(int)` and `withdraw(int)`, a class named `DepositTask` to add an amount to the balance, a class named `WithdrawTask` to withdraw an amount from the balance, and a main class that creates and launches two threads.

The program creates and submits the deposit task (line 10) and the withdraw task (line 11). The deposit task is purposely put to sleep (line 23) to let the withdraw task run. When there are not enough funds to withdraw, the withdraw task waits (line 59) for notification of the balance change from the deposit task (line 83).

A lock is created in line 44. A condition named `newDeposit` on the lock is created in line 47. A condition is bound to a lock. Before waiting or signaling the condition, a thread must first acquire the lock for the condition. The withdraw task acquires the lock in line 56, waits for the `newDeposit` condition (line 60) when there is not a sufficient amount to withdraw, and releases the lock in line 71. The deposit task acquires the lock in line 76 and signals all waiting threads (line 83) for the `newDeposit` condition after a new deposit is made.

What will happen if you replace the `while` loop in lines 58–61 with the following `if` statement?

```
if (balance < amount) {
  System.out.println("\t\t\tWait for a deposit");
  newDeposit.await();
}
```

The deposit task will notify the withdraw task whenever the balance changes. `(balance < amount)` may still be true when the withdraw task is awakened. Using the `if` statement would lead to incorrect withdraw. Using the loop statement, the withdraw task will have a chance to recheck the condition.

ever-waiting threads

Caution

Once a thread invokes `await()` on a condition, the thread waits for a signal to resume. If you forget to call `signal()` or `signalAll()` on the condition, the thread will wait forever.

`IllegalMonitorStateException`

Caution

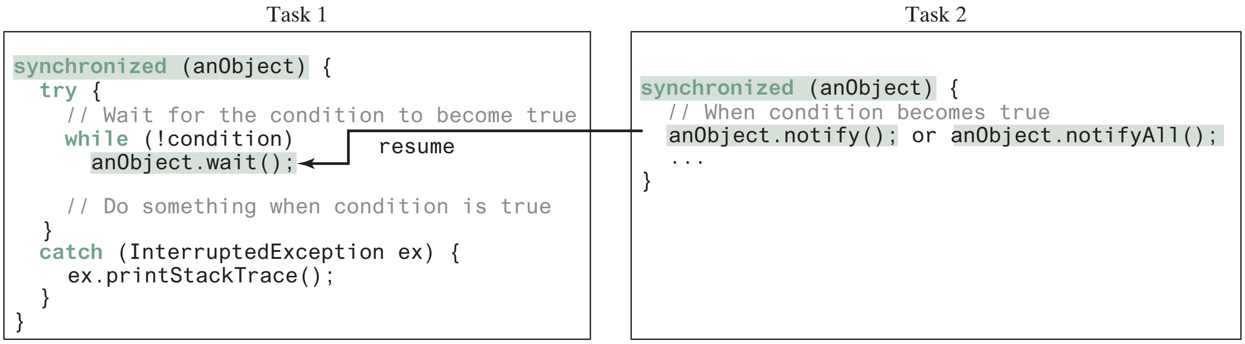
A condition is created from a `Lock` object. To invoke its method (e.g., `await()`, `signal()`, and `signalAll()`), you must first own the lock. If you invoke these methods without acquiring the lock, an `IllegalMonitorStateException` will be thrown.

Locks and conditions were introduced in Java 5. Prior to Java 5, thread communications were programmed using the object's built-in monitors. Locks and conditions are more powerful and flexible than the built-in monitor, so will not need to use monitors. However, if you are working with legacy Java code, you may encounter Java's built-in monitor.

Java's built-in monitor

monitor

A *monitor* is an object with mutual exclusion and synchronization capabilities. Only one thread can execute a method at a time in the monitor. A thread enters the monitor by acquiring a lock on it and exits by releasing the lock. *Any object can be a monitor*. An object becomes a monitor once a thread locks it. Locking is implemented using the `synchronized` keyword on a method or a block. A thread must acquire a lock before executing a synchronized method or block. A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor. You can invoke the `wait()` method on the monitor object to release the lock so some other thread can get in the monitor and perhaps change the monitor's state. When the condition is right, the other thread can invoke the `notify()` or `notifyAll()` method to signal one or all waiting threads to regain the lock and resume execution. The template for invoking these methods is shown in Figure 32.17.



```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

    // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or anObject.notifyAll();
  ...
}
```

Figure 32.17 The `wait()`, `notify()`, and `notifyAll()` methods coordinate thread communication.

The `wait()`, `notify()`, and `notifyAll()` methods must be called in a synchronized method or a synchronized block on the receiving object of these methods. Otherwise, an `IllegalMonitorStateException` will occur.

When `wait()` is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.

The `wait()`, `notify()`, and `notifyAll()` methods on an object are analogous to the `await()`, `signal()`, and `signalAll()` methods on a condition.

Check Point

32.9.1 How do you create a condition on a lock? What are the `await()`, `signal()`, and `signalAll()` methods for?

32.9.2 What would happen if the `while` loop in line 58 of Listing 32.6 was changed to an `if` statement?

`while (balance < amount)` Replaced by → `if (balance < amount)`

32.9.3 Why does the following class have a syntax error?

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() throws InterruptedException {
    Thread thread = new Thread(this);
    thread.sleep(1000);
  }

  public synchronized void run() {
  }
}
```

32.9.4 What is a possible cause for `IllegalMonitorStateException`?

32.9.5 Can `wait()`, `notify()`, and `notifyAll()` be invoked from any object? What is the purpose of these methods?

32.9.6 What is wrong in the following code?

```
synchronized (object1) {
  try {
    while (!condition) object2.wait();
  }
  catch (InterruptedException ex) {
  }
}
```

32.10 Case Study: Producer/Consumer

Key Point

This section gives the classic Consumer/Producer example for demonstrating thread coordination.

Suppose that you use a buffer to store integers and that the buffer size is limited. The buffer provides the method `write(int)` to add an `int` value to the buffer and the method `read()` to read and delete an `int` value from the buffer. To synchronize the operations, use a lock with two conditions: `notEmpty` (i.e., the buffer is not empty) and `notFull` (i.e., the buffer is not full). When a task adds an `int` to the buffer, if the buffer is full, the task will wait for the `notFull` condition. When a task reads an `int` from the buffer, if the buffer is empty, the task will wait for the `notEmpty` condition. The interaction between the two tasks is shown in Figure 32.18.

Listing 32.7 presents the complete program. The program contains the `Buffer` class (lines 50–101) and two tasks for repeatedly adding and consuming numbers to and from the buffer (lines 16–47). The `write(int)` method (lines 62–79) adds an integer to the buffer. The `read()` method (lines 81–100) deletes and returns an integer from the buffer.

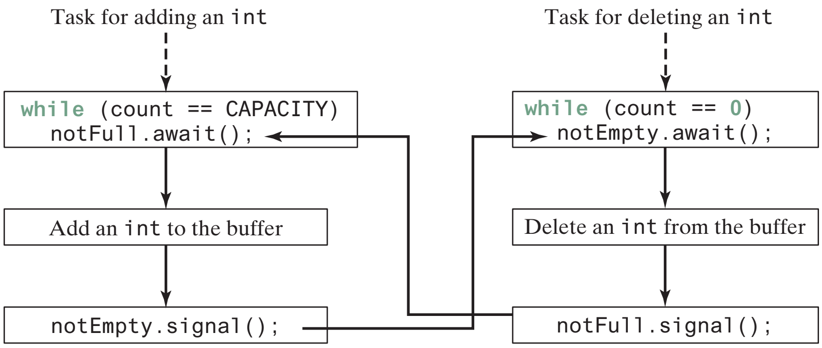

Task for adding an int
while (count == CAPACITY)
notFull.await();
Add an int to the buffer
notEmpty.signal();
Task for deleting an int
while (count == 0)
notEmpty.await();
Delete an int from the buffer
notFull.signal();

FIGURE 32.18 The conditions `notFull` and `notEmpty` are used to coordinate task interactions.

The buffer is actually a first-in, first-out queue (lines 52 and 53). The conditions `notEmpty` and `notFull` on the lock are created in lines 59 and 60. The conditions are bound to a lock. A lock must be acquired before a condition can be applied. If you use the `wait()` and `notify()` methods to rewrite this example, you have to designate two objects as monitors.

LISTING 32.7 ConsumerProducer.java

create a buffer

create two threads

producer task

consumer task

```
 1 import java.util.concurrent.*;
 2 import java.util.concurrent.locks.*;
 3
 4 public class ConsumerProducer {
 5   private static Buffer buffer = new Buffer();
 6
 7   public static void main(String[] args) {
 8     // Create a thread pool with two threads
 9     ExecutorService executor = Executors.newFixedThreadPool(2);
10     executor.execute(new ProducerTask());
11     executor.execute(new ConsumerTask());
12     executor.shutdown();
13   }
14
15   // A task for adding an int to the buffer
16   private static class ProducerTask implements Runnable {
17     public void run() {
18       try {
19         int i = 1;
20         while (true) {
21           System.out.println("Producer writes " + i);
22           buffer.write(i++); // Add a value to the buffer
23           // Put the thread into sleep
24           Thread.sleep((int)(Math.random() * 10000));
25         }
26       }
27       catch (InterruptedException ex) {
28         ex.printStackTrace();
29       }
30     }
31   }
32
33   // A task for reading and deleting an int from the buffer
34   private static class ConsumerTask implements Runnable {
35     public void run() {
      try {
        while (true) {
          System.out.println("\t\t\tConsumer reads " + buffer.read());
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

  // An inner class for buffer
  private static class Buffer {
    private static final int CAPACITY = 1; // buffer size
    private java.util.LinkedList<Integer> queue =
      new java.util.LinkedList<>();

    // Create a new lock
    private static Lock lock = new ReentrantLock();

    // Create two conditions
    private static Condition notEmpty = lock.newCondition();
    private static Condition notFull = lock.newCondition();

    public void write(int value) {
      lock.lock(); // Acquire the lock
      try {
        while (queue.size() == CAPACITY) {
          System.out.println("Wait for notFull condition");
          notFull.await();
        }

        queue.offer(value);
        notEmpty.signal(); // Signal notEmpty condition
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
      finally {
        lock.unlock(); // Release the lock
      }
    }

    public int read() {
      int value = 0;
      lock.lock(); // Acquire the lock
      try {
        while (queue.isEmpty()) {
          System.out.println("\t\t\tWait for notEmpty condition");
          notEmpty.await();
        }

        value = queue.remove();
        notFull.signal(); // Signal notFull condition
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
```

create a lock

create a condition

create a condition

acquire the lock

wait for `notFull`

signal `notEmpty`

release the lock

acquire the lock

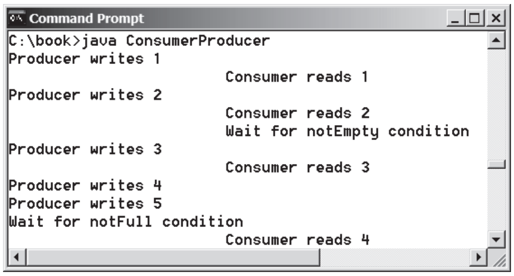
wait for `notEmpty`

signal `notFull`

release the lock

```
      finally {
        lock.unlock(); // Release the lock
        return value;
      }
    }
  }
}
```

A sample run of the program is shown in Figure 32.19.


Command Prompt

```
C:\book>java ConsumerProducer
Producer writes 1
                                Consumer reads 1
Producer writes 2
                                Consumer reads 2
                                Wait for notEmpty condition
Producer writes 3
                                Consumer reads 3
Producer writes 4
Producer writes 5
Wait for notFull condition
                                Consumer reads 4
```

FIGURE 32.19 Locks and conditions are used for communications between the Producer and Consumer threads.

Check Point

32.10.1 Can the `read` and `write` methods in the `Buffer` class be executed concurrently?

32.10.2 When invoking the `read` method, what happens if the queue is empty?

32.10.3 When invoking the `write` method, what happens if the queue is full?

32.11 Blocking Queues

Key Point

Java Collections Framework provides `ArrayBlockingQueue`, `LinkedBlockingQueue`, *and* `PriorityBlockingQueue` *for supporting blocking queues.*

blocking queue

Queues and priority queues were introduced in Section 20.9. A *blocking queue* causes a thread to block when you try to add an element to a full queue or to remove an element from an empty queue. The `BlockingQueue` interface extends `java.util.Queue` and provides the synchronized `put` and `take` methods for adding an element to the tail of the queue and for removing an element from the head of the queue, as shown in Figure 32.20.

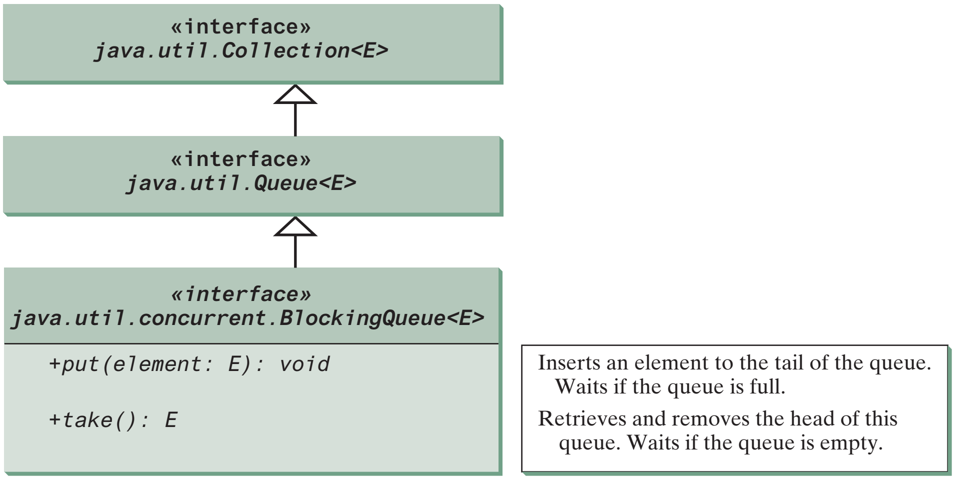

«interface»
java.util.Collection<E>
«interface»
java.util.Queue<E>
«interface»
java.util.concurrent.BlockingQueue<E>
+put(element: E): void
+take(): E
Inserts an element to the tail of the queue. Waits if the queue is full.
Retrieves and removes the head of this queue. Waits if the queue is empty.

FIGURE 32.20 `BlockingQueue` is a subinterface of `Queue`.

Three concrete blocking queues—`ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue`—are provided in Java, as shown in Figure 32.21. All are in the `java.util.concurrent` package. `ArrayBlockingQueue` implements a blocking queue using an array. You have to specify a capacity or an optional fairness to construct an `ArrayBlockingQueue`. `LinkedBlockingQueue` implements a blocking queue using a linked list. You can create an unbounded or bounded `LinkedBlockingQueue`. `PriorityBlockingQueue` is a priority queue. You can create an unbounded or bounded priority queue.

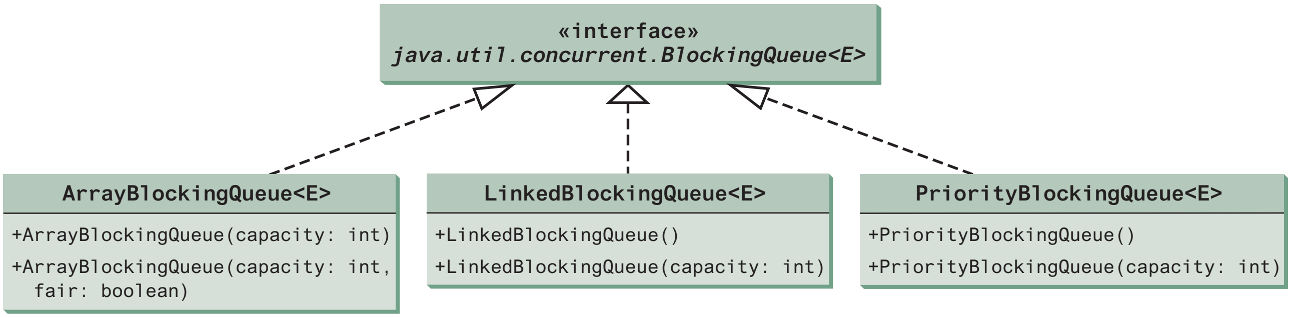

«interface»
java.util.concurrent.BlockingQueue<E>
ArrayBlockingQueue<E>
+ArrayBlockingQueue(capacity: int)
+ArrayBlockingQueue(capacity: int, fair: boolean)
LinkedBlockingQueue<E>
+LinkedBlockingQueue()
+LinkedBlockingQueue(capacity: int)
PriorityBlockingQueue<E>
+PriorityBlockingQueue()
+PriorityBlockingQueue(capacity: int)

FIGURE 32.21 `ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue` are concrete blocking queues.

Note

The `put` method will never block an unbounded `LinkedBlockingQueue` or `PriorityBlockingQueue`.

unbounded queue

Listing 32.8 gives an example of using an `ArrayBlockingQueue` to simplify the Consumer/Producer example in Listing 32.10. Line 5 creates an `ArrayBlockingQueue` to store integers. The Producer thread puts an integer into the queue (line 22) and the Consumer thread takes an integer from the queue (line 38).

LISTING 32.8 ConsumerProducerUsingBlockingQueue.java

```
 1  import java.util.concurrent.*;
 2
 3  public class ConsumerProducerUsingBlockingQueue {
 4    private static ArrayBlockingQueue<Integer> buffer =
 5      new ArrayBlockingQueue<>(2);
 6
 7    public static void main(String[] args) {
 8      // Create a thread pool with two threads
 9      ExecutorService executor = Executors.newFixedThreadPool(2);
10      executor.execute(new ProducerTask());
11      executor.execute(new ConsumerTask());
12      executor.shutdown();
13    }
14
15    // A task for adding an int to the buffer
16    private static class ProducerTask implements Runnable {
17      public void run() {
18        try {
19          int i = 1;
20          while (true) {
21            System.out.println("Producer writes " + i);
22            buffer.put(i++); // Add any value to the buffer, say, 1
23            // Put the thread into sleep
```

create a buffer

create two threads

producer task

put

Consumer task

take

```
          Thread.sleep((int)(Math.random() * 10000));
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

  // A task for reading and deleting an int from the buffer
  private static class ConsumerTask implements Runnable {
    public void run() {
      try {
        while (true) {
          System.out.println("\t\t\tConsumer reads " + buffer.take());
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

In Listing 32.7, you used locks and conditions to synchronize the Producer and Consumer threads. This program does not use locks and conditions because synchronization is already implemented in `ArrayBlockingQueue`.

Check Point

32.11.1 What is a blocking queue? What blocking queues are supported in Java?

32.11.2 What method do you use to add an element to an `ArrayBlockingQueue`? What happens if the queue is full?

32.11.3 What method do you use to retrieve an element from an `ArrayBlockingQueue`? What happens if the queue is empty?

Key Point

32.12 Semaphores

Semaphores can be used to restrict the number of threads that access a shared resource.

semaphore

In computer science, a *semaphore* is an object that controls the access to a common resource. Before accessing the resource, a thread must acquire a permit from the semaphore. After finishing with the resource, the thread must return the permit back to the semaphore, as shown in Figure 32.22.

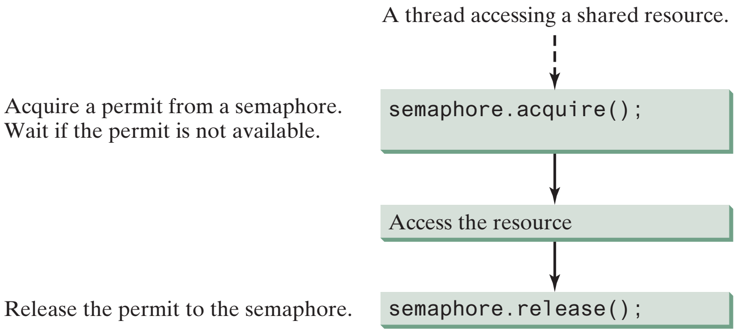
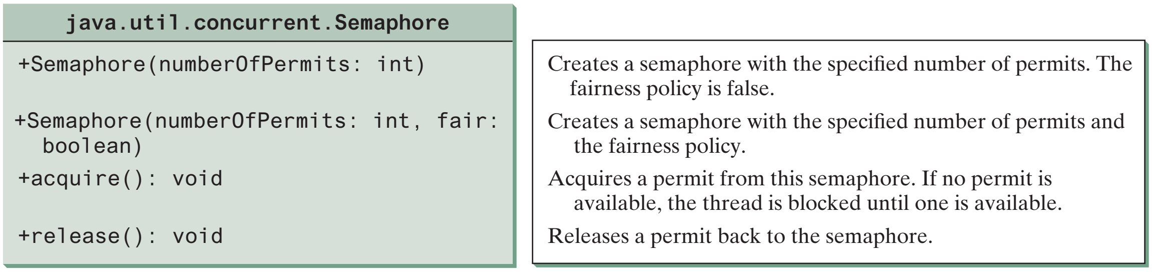

A thread accessing a shared resource.
Acquire a permit from a semaphore. Wait if the permit is not available.
semaphore.acquire();
Access the resource
Release the permit to the semaphore.
semaphore.release();

FIGURE 32.22 A limited number of threads can access a shared resource controlled by a semaphore.

To create a semaphore, you have to specify the number of permits with an optional fairness policy, as shown in Figure 32.23. A task acquires a permit by invoking the semaphore's `acquire()` method and releases the permit by invoking the semaphore's `release()` method. Once a permit is acquired, the total number of available permits in a semaphore is reduced by `1`. Once a permit is released, the total number of available permits in a semaphore is increased by `1`.



java.util.concurrent.Semaphore	
+Semaphore(numberOfPermits: int)	Creates a semaphore with the specified number of permits. The fairness policy is false.
+Semaphore(numberOfPermits: int, fair: boolean)	Creates a semaphore with the specified number of permits and the fairness policy.
+acquire(): void	Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.
+release(): void	Releases a permit back to the semaphore.

FIGURE 32.23 The `Semaphore` class contains the methods for accessing a semaphore.

A semaphore with just one permit can be used to simulate a mutually exclusive lock. Listing 32.9 revises the `Account` inner class in Listing 32.9 using a semaphore to ensure that only one thread at a time can access the `deposit` method.

LISTING 32.9 New Account Inner Class

```
 1  // An inner class for Account
 2  private static class Account {
 3    // Create a semaphore
 4    private static Semaphore semaphore = new Semaphore(1);
 5    private int balance = 0;
 6
 7    public int getBalance() {
 8      return balance;
 9    }
10
11    public void deposit(int amount) {
12      try {
13        semaphore.acquire(); // Acquire a permit
14        int newBalance = balance + amount;
15
16        // This delay is deliberately added to magnify the
17        // data-corruption problem and make it easy to see
18        Thread.sleep(5);
19
20        balance = newBalance;
21      }
22      catch (InterruptedException ex) {
23      }
24      finally {
25        semaphore.release(); // Release a permit
26      }
27    }
28  }
```

create a semaphore

acquire a permit

release a permit

A semaphore with one permit is created in line 4. A thread first acquires a permit when executing the deposit method in line 13. After the balance is updated, the thread releases the permit in line 25. It is a good practice to always place the `release()` method in the `finally` clause to ensure that the permit is finally released even in the case of exceptions.

32.12.1 What are the similarities and differences between a lock and a semaphore?

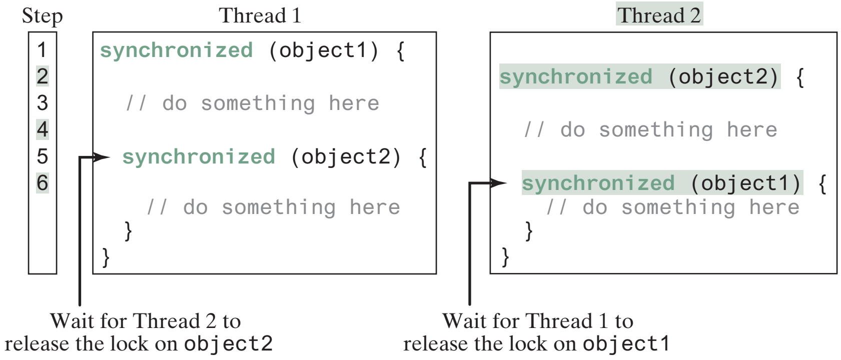
32.12.2 How do you create a semaphore that allows three concurrent threads? How do you acquire a semaphore? How do you release a semaphore?

32.13 Avoiding Deadlocks

Deadlocks can be avoided by using a proper resource ordering.

deadlock

Sometimes two or more threads need to acquire the locks on several shared objects. This could cause a *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object. Consider the scenario with two threads and two objects, as shown in Figure 32.24. Thread 1 has acquired a lock on `object1`, and Thread 2 has acquired a lock on `object2`. Now Thread 1 is waiting for the lock on `object2`, and Thread 2 for the lock on `object1`. Each thread waits for the other to release the lock it needs and until that happens, neither can continue to run.



```
synchronized (object1) {

  // do something here

  synchronized (object2) {

    // do something here
  }
}
```

```
synchronized (object2) {

  // do something here

  synchronized (object1) {
    // do something here
  }
}
```

Wait for Thread 2 to release the lock on `object2`

Wait for Thread 1 to release the lock on `object1`

Figure 32.24 Thread 1 and Thread 2 are deadlocked.

resource ordering

Deadlock is easily avoided by using a simple technique known as *resource ordering*. With this technique, you assign an order to all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order. For example in Figure 32.24, suppose the objects are ordered as `object1` and `object2`. Using the resource-ordering technique, Thread 2 must acquire a lock on `object1` first, then on `object2`. Once Thread 1 acquires a lock on `object1`, Thread 2 has to wait for a lock on `object1`. Thus, Thread 1 will be able to acquire a lock on `object2` and no deadlock will occur.

32.13.1 What is a deadlock? How can you avoid deadlock?

32.14 Thread States

A thread state indicates the status of thread.

Tasks are executed in threads. Threads can be in one of the five states: New, Ready, Running, Blocked, or Finished (see Figure 32.25).

When a thread is newly created, it enters the ***New*** state. After a thread is started by calling its `start()` method, it enters the ***Ready*** state. A ready thread is runnable but may not be running yet. The operating system has to allocate CPU time to it.

When a ready thread begins executing, it enters the ***Running*** state. A running thread can enter the ***Ready*** state if its given CPU time expires or its `yield()` method is called.

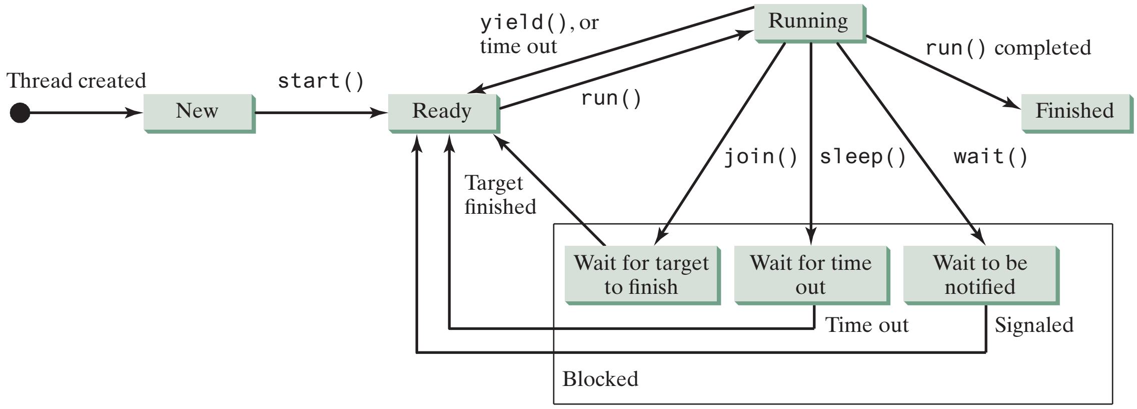


Figure 32.25 A thread can be in one of the five states: New, Ready, Running, Blocked, or Finished.

A thread can enter the ***Blocked*** state (i.e., become inactive) for several reasons. It may have invoked the `join()`, `sleep()`, or `wait()` method. It may be waiting for an I/O operation to finish. A blocked thread may be reactivated when the action inactivating it is reversed. For example, if a thread has been put to sleep and the sleep time has expired, the thread is reactivated and enters the **Ready** state.

Finally, a thread is ***Finished*** if it completes the execution of its `run()` method.

The `isAlive()` method is used to find out the state of a thread. It returns `true` if a thread is in the **Ready**, **Blocked**, or **Running** state; it returns `false` if a thread is new and has not started or if it is finished.

The `interrupt()` method interrupts a thread in the following way: If a thread is currently in the **Ready** or **Running** state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the **Ready** state, and a `java.lang.InterruptedException` is thrown.

Check Point

32.14.1 What is a thread state? Describe the states for a thread.

32.15 Synchronized Collections

Key Point

Java Collections Framework provides synchronized collections for lists, sets, and maps.

synchronized collection

The classes in the Java Collections Framework are not thread-safe; that is, their contents may become corrupted if they are accessed and updated concurrently by multiple threads. You can protect the data in a collection by locking the collection or by using synchronized collections.

synchronization wrapper

The `Collections` class provides six static methods for wrapping a collection into a synchronized version, as shown in Figure 32.26. The collections created using these methods are called *synchronization wrappers*.

`java.util.Collections`	
`+synchronizedCollection(c: Collection): Collection`	Returns a synchronized collection.
`+synchronizedList(list: List): List`	Returns a synchronized list from the specified list.
`+synchronizedMap(m: Map): Map`	Returns a synchronized map from the specified map.
`+synchronizedSet(s: Set): Set`	Returns a synchronized set from the specified set.
`+synchronizedSortedMap(s: SortedMap): SortedMap`	Returns a synchronized sorted map from the specified sorted map.
`+synchronizedSortedSet(s: SortedSet): SortedSet`	Returns a synchronized sorted set.

Figure 32.26 You can obtain synchronized collections using the methods in the `Collections` class.

Invoking `synchronizedCollection(Collection c)` returns a new `Collection` object, in which all the methods that access and update the original collection `c` are synchronized. These methods are implemented using the `synchronized` keyword. For example, the `add` method is implemented like this:

```
public boolean add(E o) {
  synchronized (this) {
    return c.add(o);
  }
}
```

Synchronized collections can be safely accessed and modified by multiple threads concurrently.

Note

The methods in `java.util.Vector`, `java.util.Stack`, and `java.util.Hashtable` are already synchronized. These are old classes introduced in JDK 1.0. Starting with JDK 1.5, you should use `java.util.ArrayList` to replace `Vector`, `java.util.LinkedList` to replace `Stack`, and `java.util.Map` to replace `Hashtable`. If synchronization is needed, use a synchronization wrapper.

fail-fast

The synchronization wrapper classes are thread-safe, but the iterator is *fail-fast*. This means that if you are using an iterator to traverse a collection while the underlying collection is being modified by another thread, then the iterator will immediately fail by throwing `java.util.ConcurrentModificationException`, which is a subclass of `RuntimeException`. To avoid this error, you need to create a synchronized collection object and acquire a lock on the object when traversing it. For example, to traverse a set, you have to write the code like this:

```
Set hashSet = Collections.synchronizedSet(new HashSet());

synchronized (hashSet) { // Must synchronize it
  Iterator iterator = hashSet.iterator();

  while (iterator.hasNext()) {
    System.out.println(iterator.next());
  }
}
```

Failure to do so may result in nondeterministic behavior, such as a `ConcurrentModificationException`.

Check Point

32.15.1 What is a synchronized collection? Is `ArrayList` synchronized? How do you make it synchronized?

32.15.2 Explain why an iterator is fail-fast.

32.16 Parallel Programming

Key Point

The Fork/Join Framework is used for parallel programming in Java.

Section 7.12 introduced the `Arrays.sort` and `Arrays.parallelSort` method for sorting an array. The `parallelSort` method utilizes multiple processors to reduce sort time. Chapter 22 introduced parallel streams for executing stream operations in parallel to speed up processing using multiple processors. The parallel processing are implemented using the Fork/Join Framework. This section, introduces the new Fork/Join Framework so you can write own code for parallel programming.

Fork/Join Framework

JDK 7 feature

The *Fork/Join Framework* is illustrated in Figure 32.27 (the diagram resembles a fork, hence its name). A problem is divided into nonoverlapping subproblems, which can be solved independently in parallel. The solutions to all subproblems are then joined to obtain an overall solution for the problem. This is the parallel implementation of the divide-and-conquer approach. In JDK 7's Fork/Join Framework, a *fork* can be viewed as an independent task that runs on a thread.

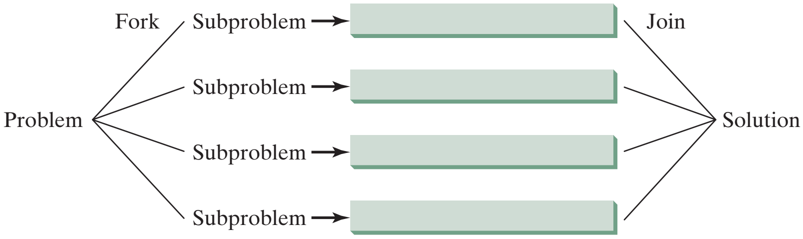

Fork
Join
Problem
Subproblem
Subproblem
Subproblem
Subproblem
Solution

FIGURE 32.27 The nonoverlapping subproblems are solved in parallel.

The framework defines a task using the `ForkJoinTask` class, as shown in Figure 32.28 and executes a task in an instance of `ForkJoinPool`, as shown in Figure 32.29.

`ForkJoinTask`
`ForkJoinPool`

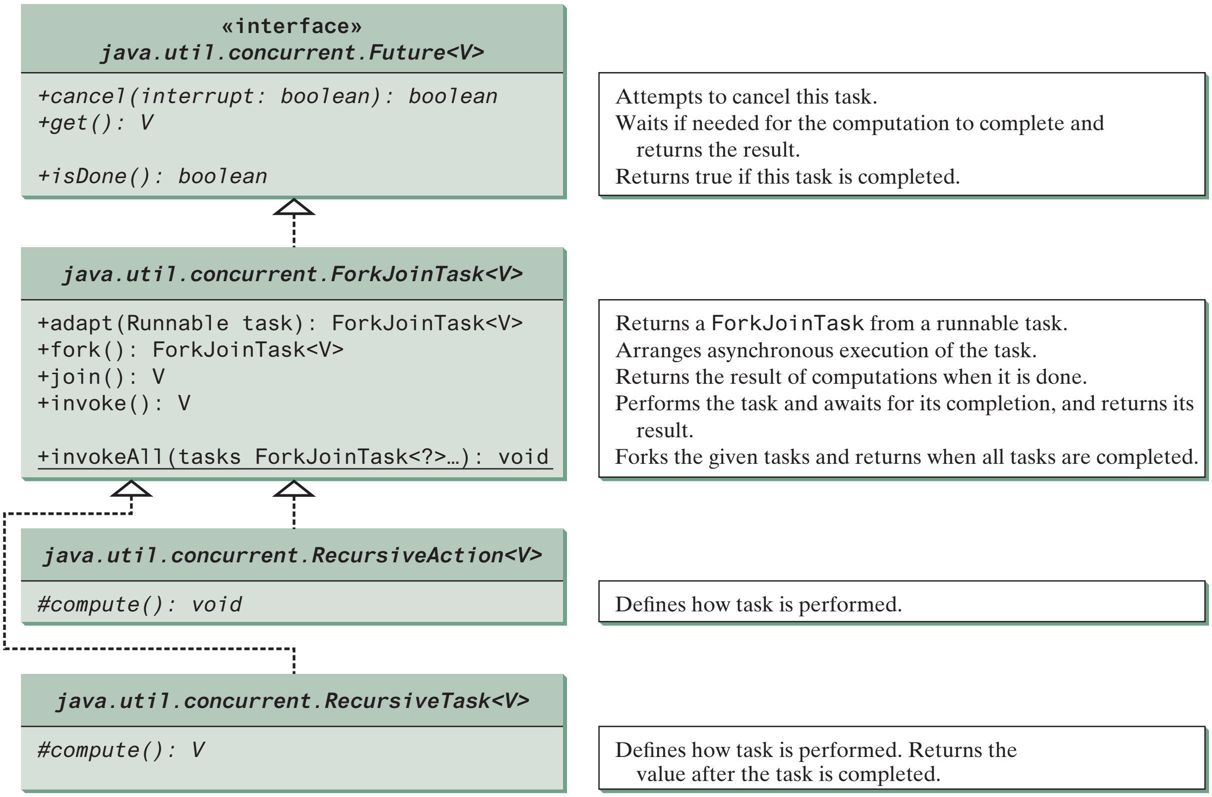

«interface»
java.util.concurrent.Future<V>
+cancel(interrupt: boolean): boolean
+get(): V
+isDone(): boolean
Attempts to cancel this task.
Waits if needed for the computation to complete and returns the result.
Returns true if this task is completed.
java.util.concurrent.ForkJoinTask<V>
+adapt(Runnable task): ForkJoinTask<V>
+fork(): ForkJoinTask<V>
+join(): V
+invoke(): V
+invokeAll(tasks ForkJoinTask<?>…): void
Returns a ForkJoinTask from a runnable task.
Arranges asynchronous execution of the task.
Returns the result of computations when it is done.
Performs the task and awaits for its completion, and returns its result.
Forks the given tasks and returns when all tasks are completed.
java.util.concurrent.RecursiveAction<V>
#compute(): void
Defines how task is performed.
java.util.concurrent.RecursiveTask<V>
#compute(): V
Defines how task is performed. Returns the value after the task is completed.

FIGURE 32.28 The `ForkJoinTask` class defines a task for asynchronous execution.

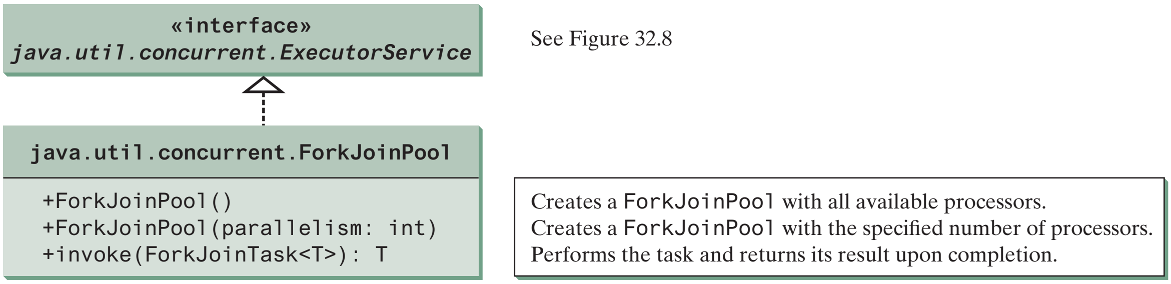

«interface»
java.util.concurrent.ExecutorService
See Figure 32.8
java.util.concurrent.ForkJoinPool
+ForkJoinPool()
+ForkJoinPool(parallelism: int)
+invoke(ForkJoinTask<T>): T
Creates a ForkJoinPool with all available processors.
Creates a ForkJoinPool with the specified number of processors.
Performs the task and returns its result upon completion.

FIGURE 32.29 The `ForkJoinPool` executes Fork/Join tasks.

`ForkJoinTask` is the abstract base class for tasks. A `ForkJoinTask` is a thread-like entity, but it is much lighter than a normal thread because huge numbers of tasks and subtasks can be executed by a small number of actual threads in a `ForkJoinPool`. The tasks are primarily coordinated using `fork()` and `join()`. Invoking `fork()` on a task arranges asynchronous execution and invoking `join()` waits until the task is completed. The `invoke()` and `invokeAll(tasks)` methods implicitly invoke `fork()` to execute the task and `join()` to wait for the tasks to complete and return the result, if any. Note the static method `invokeAll` takes a variable number of `ForkJoinTask` arguments using the `...` syntax, which is introduced in Section 7.9.

The Fork/Join Framework is designed to parallelize divide-and-conquer solutions, which are naturally recursive. `RecursiveAction` and `RecursiveTask` are two subclasses of `ForkJoinTask`. To define a concrete task class, your class should extend `RecursiveAction` or `RecursiveTask`. `RecursiveAction` is for a task that doesn't return a value and `RecursiveTask` is for a task that does return a value. Your task class should override the `compute()` method to specify how a task is performed.

`RecursiveAction`
`RecursiveTask`

We now use a merge sort to demonstrate how to develop parallel programs using the Fork/Join Framework. The merge sort algorithm (introduced in Section 25.3) divides the array into two halves and applies a merge sort on each half recursively. After the two halves are sorted, the algorithm merges them. Listing 32.10 gives a parallel implementation of the merge sort algorithm and compares its execution time with a sequential sort.

LISTING 32.10 `ParallelMergeSort.java`

invoke parallel sort
invoke sequential sort
create a `ForkJoinTask`
create a `ForkJoinPool`
execute a task
define concrete `ForkJoinTask`

```java
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.ForkJoinPool;

public class ParallelMergeSort {
  public static void main(String[] args) {
    final int SIZE = 7000000;
    int[] list1 = new int[SIZE];
    int[] list2 = new int[SIZE];

    for (int i = 0; i < list1.length; i++)
      list1[i] = list2[i] = (int)(Math.random() * 10000000);

    long startTime = System.currentTimeMillis();
    parallelMergeSort(list1); // Invoke parallel merge sort
    long endTime = System.currentTimeMillis();
    System.out.println("\nParallel time with "
      + Runtime.getRuntime().availableProcessors() +
     " processors is " + (endTime - startTime) + " milliseconds");

    startTime = System.currentTimeMillis();
    MergeSort.mergeSort(list2); // MergeSort is in Listing 23.5
    endTime = System.currentTimeMillis();
    System.out.println("\nSequential time is " +
      (endTime - startTime) + " milliseconds");
  }

  public static void parallelMergeSort(int[] list) {
    RecursiveAction mainTask = new SortTask(list);
    ForkJoinPool pool = new ForkJoinPool();
    pool.invoke(mainTask);
  }

  private static class SortTask extends RecursiveAction {
    private final int THRESHOLD = 500;
35      private int[] list;
36
37      SortTask(int[] list) {
38        this.list = list;
39      }
40
41      @Override
42      protected void compute() {
43        if (list.length < THRESHOLD)
44          java.util.Arrays.sort(list);
45        else {
46          // Obtain the first half
47          int[] firstHalf = new int[list.length / 2];
48          System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
49
50          // Obtain the second half
51          int secondHalfLength = list.length - list.length / 2;
52          int[] secondHalf = new int[secondHalfLength];
53          System.arraycopy(list, list.length / 2,
54            secondHalf, 0, secondHalfLength);
55
56          // Recursively sort the two halves
57          invokeAll(new SortTask(firstHalf),
58            new SortTask(secondHalf));
59
60          // Merge firstHalf with secondHalf into list
61          MergeSort.merge(firstHalf, secondHalf, list);
62        }
63      }
64    }
65  }
```

perform the task

sort a small list

split into two parts

solve each part

merge two parts

```
Parallel time with two processors is 2829 milliseconds
Sequential time is 4751 milliseconds
```

Since the sort algorithm does not return a value, we define a concrete `ForkJoinTask` class by extending `RecursiveAction` (lines 33–64). The `compute` method is overridden to implement a recursive merge sort (lines 42–63). If the list is small, it is more efficient to be solved sequentially (line 44). For a large list, it is split into two halves (lines 47–54). The two halves are sorted concurrently (lines 57 and 58) and then merged (line 61).

The program creates a main `ForkJoinTask` (line 28), a `ForkJoinPool` (line 29), and places the main task for execution in a `ForkJoinPool` (line 30). The `invoke` method will return after the main task is completed.

When executing the main task, the task is split into subtasks, and the subtasks are invoked using the `invokeAll` method (lines 57 and 58). The `invokeAll` method will return after all the subtasks are completed. Note each subtask is further split into smaller tasks recursively. Huge numbers of subtasks may be created and executed in the pool. The Fork/Join Framework automatically executes and coordinates all the tasks efficiently.

The `MergeSort` class is defined in Listing 23.5. The program invokes `MergeSort.merge` to merge two sorted sublists (line 61). The program also invokes `MergeSort.mergeSort` (line 21) to sort a list using merge sort sequentially. You can see that the parallel sort is much faster than the sequential sort.

Note the loop for initializing the list can also be parallelized. However, you should avoid using `Math.random()` in the code because it is synchronized and cannot be executed in parallel (see Programming Exercise 32.12). The `parallelMergeSort` method only sorts an array of `int` values, but you can modify it to become a generic method (see Programming Exercise 32.13).

In general, a problem can be solved in parallel using the following pattern:

```
if (the program is small)
  solve it sequentially;
else {
  divide the problem into nonoverlapping subproblems;
  solve the subproblems concurrently;
  combine the results from subproblems to solve the whole problem;
}
```

Listing 32.11 develops a parallel method that finds the maximal number in a list.

LISTING 32.11 ParallelMax.java

invoke `max`

create a `ForkJoinTask`

create a `ForkJoinPool`

execute a task

define concrete `ForkJoinTask`

perform the task

solve a small problem

```java
import java.util.concurrent.*;

public class ParallelMax {
  public static void main(String[] args) {
    // Create a list
    final int N = 9000000;
    int[] list = new int[N];
    for (int i = 0; i < list.length; i++)
      list[i] = i;

    long startTime = System.currentTimeMillis();
    System.out.println("\nThe maximal number is " + max(list));
    long endTime = System.currentTimeMillis();
    System.out.println("The number of processors is " +
      Runtime.getRuntime().availableProcessors());
    System.out.println("Time is " + (endTime - startTime)
      + " milliseconds");
  }

  public static int max(int[] list) {
    RecursiveTask<Integer> task = new MaxTask(list, 0, list.length);
    ForkJoinPool pool = new ForkJoinPool();
    return pool.invoke(task);
  }

  private static class MaxTask extends RecursiveTask<Integer> {
    private final static int THRESHOLD = 1000;
    private int[] list;
    private int low;
    private int high;

    public MaxTask(int[] list, int low, int high) {
      this.list = list;
      this.low = low;
      this.high = high;
    }

    @Override
    public Integer compute() {
      if (high - low < THRESHOLD) {
        int max = list[0];
        for (int i = low; i < high; i++)
          if (list[i] > max)
            max = list[i];
        return new Integer(max);
46          }
47          else {
48            int mid = (low + high) / 2;
49            RecursiveTask<Integer> left = new MaxTask(list, low, mid);
50            RecursiveTask<Integer> right = new MaxTask(list, mid, high);
51
52            right.fork();
53            left.fork();
54            return new Integer(Math.max(left.join().intValue(),
55              right.join().intValue()));
56          }
57        }
58      }
59    }
```

split into two parts

fork right

fork left

join tasks

```
The maximal number is 8999999
The number of processors is 2
Time is 44 milliseconds
```

Since the algorithm returns an integer, we define a task class for fork join by extending `RecursiveTask<Integer>` (lines 26–58). The `compute` method is overridden to return the max element in a `list[low..high]` (lines 39–57). If the list is small, it is more efficient to be solved sequentially (lines 40–46). For a large list, it is split into two halves (lines 48–50). The tasks `left` and `right` find the maximal element in the left half and right half, respectively. Invoking `fork()` on the task causes the task to be executed (lines 52 and 53). The `join()` method awaits for the task to complete and then returns the result (lines 54 and 55).

Check Point

32.16.1 How do you define a `ForkJoinTask`? What are the differences between `RecursiveAction` and `RecursiveTask`?

32.16.2 How do you tell the system to execute a task?

32.16.3 What method can you use to test if a task has been completed?

32.16.4 How do you create a `ForkJoinPool`? How do you place a task into a `ForkJoinPool`?

KEY TERMS

condition	32-18
deadlock	32-30
fail-fast	32-32
fairness policy	32-17
Fork/Join Framework	32-32
lock	32-16
monitor	32-22
multithreading	32-2
race condition	32-15
semaphore	32-28
synchronization wrapper	32-31
synchronized block	32-16
thread	32-2
thread-safe	32-15

CHAPTER SUMMARY

1. Each task is an instance of the `Runnable` interface. A *thread* is an object that facilitates the execution of a task. You can define a task class by implementing the `Runnable` interface and create a thread by wrapping a task using a `Thread` constructor.

2. After a thread object is created, use the `start()` method to start a thread, and the `sleep(long)` method to put a thread to sleep so other threads get a chance to run.

3. A thread object never directly invokes the `run` method. The JVM invokes the `run` method when it is time to execute the thread. Your class must override the `run` method to tell the system what the thread will do when it runs.

4. To prevent threads from corrupting a shared resource, use *synchronized* methods or blocks. A *synchronized method* acquires a *lock* before it executes. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static method, the lock is on the class.

5. A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block of the code in a method. This block is referred to as a *synchronized block*.

6. You can use explicit locks and *conditions* to facilitate communications among threads, as well as using the built-in monitor for objects.

7. The blocking queues (`ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue`) provided in the Java Collections Framework provide automatical synchronization for the access to a queue.

8. You can use semaphores to restrict the number of concurrent accesses to a shared resource.

9. *Deadlock* occurs when two or more threads acquire locks on multiple objects and each has a lock on one object and is waiting for the lock on the other object. The *resource-ordering technique* can be used to avoid deadlock.

10. The JDK 7's Fork/Join Framework is designed for developing parallel programs. You can define a task class that extends `RecursiveAction` or `RecursiveTask` and execute the tasks concurrently in `ForkJoinPool` and obtain the overall solution after all tasks are completed.

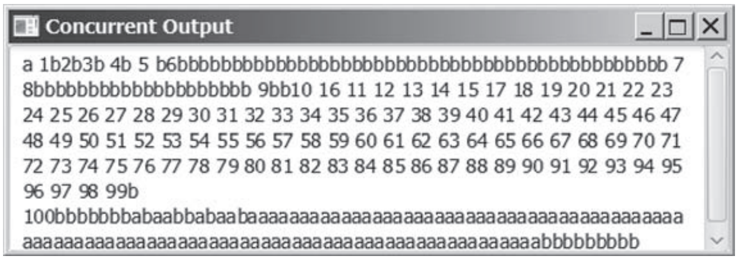
Quiz

Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

Sections 32.1–32.5

***32.1** (*Revise Listing 32.1*) Rewrite Listing 32.1 to display the output in a text area, as shown in Figure 32.30.



```
a 1b2b3b 4b 5 b6bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb 7
8bbbbbbbbbbbbbbbbbbbb 9bb10 16 11 12 13 14 15 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99b
100bbbbbbabaabbabaa baaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb
```

Figure 32.30 The output from three threads is displayed in a text area.

32.2 (*Racing cars*) Rewrite Programming Exercise 15.29 using a thread to control car racing. Compare the program with Programming Exercise 15.29 by setting the delay time to 10 in both the programs. Which one runs the animation faster?

32.3 (*Raise flags*) Rewrite Listing 15.13 using a thread to animate a flag being raised. Compare the program with Listing 15.13 by setting the delay time to 10 in both programs. Which one runs the animation faster?

Sections 32.8–32.12

32.4 (*Synchronize threads*) Write a program that launches 1,000 threads. Each thread adds `1` to a variable `sum` that initially is `0`. You need to pass `sum` by reference to each thread. In order to pass it by reference, define an `Integer` wrapper object to hold `sum`. Run the program with and without synchronization to see its effect.

32.5 (*Display a running fan*) Rewrite Programming Exercise 15.28 using a thread to control the fan animation.

32.6 (*Bouncing balls*) Rewrite Listing 15.17, BallPane.java using a thread to animate bouncing ball movements.

32.7 (*Control a clock*) Rewrite Programming Exercise 15.32 using a thread to control the clock animation.

32.8 (*Account synchronization*) Rewrite Listing 32.6, ThreadCooperation.java, using the object's `wait()` and `notifyAll()` methods.

32.9 (*Demonstrate* `ConcurrentModificationException`) The iterator is *fail-fast*. Write a program to demonstrate it by creating two threads that concurrently access and modify a set. The first thread creates a hash set filled with numbers and adds a new number to the set every second. The second thread obtains an iterator for the set and traverses the set back and forth through the iterator every second. You will receive a `ConcurrentModificationException` because the underlying set is being modified in the first thread while the set in the second thread is being traversed.

***32.10** (*Use synchronized sets*) Using synchronization, correct the problem in the preceding exercise so that the second thread does not throw a `ConcurrentModificationException`.

Section 32.15

***32.11** (*Demonstrate deadlock*) Write a program that demonstrates deadlock.

Section 32.18

***32.12** (*Parallel array initializer*) Implement the following method using the Fork/Join Framework to assign random values to the list.

```
public static void parallelAssignValues(double[] list)
```

Write a test program that creates a list with 9,000,000 elements and invokes `parallelAssignValues` to assign random values to the list. Also implement a sequential algorithm and compare the execution time of the two. Note if you use `Math.random()`, your parallel code execution time will be worse than the sequential code execution time because `Math.random()` is synchronized and cannot be executed in parallel. To fix this problem, create a `Random` object for assigning random values to a small list.

32.13 (*Generic parallel merge sort*) Revise Listing 32.10, ParallelMergeSort.java, to define a generic `parallelMergeSort` method as follows:

```
public static <E extends Comparable<E>> void
  parallelMergeSort(E[] list)
```

***32.14** (*Parallel quick sort*) Implement the following method in parallel to sort a list using quick sort (see Listing 23.7):

```
public static void parallelQuickSort(int[] list)
```

Write a test program that times the execution time for a list of size 9,000,000 using this parallel method and a sequential method.

***32.15** (*Parallel sum*) Implement the following method using Fork/Join to find the sum of a list.

```
public static double parallelSum(double[] list)
```

Write a test program that finds the sum in a list of 9,000,000 double values.

***32.16** (*Parallel matrix addition*) Programming Exercise 8.5 describes how to perform matrix addition. Suppose you have multiple processors, so you can speed up the matrix addition. Implement the following method in parallel:

```
public static double[][] parallelAddMatrix(
  double[][] a, double[][] b)
```

Write a test program that measures the execution time for adding two 2,000 × 2,000 matrices using the parallel method and sequential method, respectively.

***32.17** (*Parallel matrix multiplication*) Programming Exercise 7.6 describes how to perform matrix multiplication. Suppose that you have multiple processors, so you can speed up the matrix multiplication. Implement the following method in parallel:

```
public static double[][] parallelMultiplyMatrix(
  double[][] a, double[][] b)
```

Write a test program that measures the execution time for multiplying two 2,000 × 2,000 matrices using the parallel method and sequential method, respectively.

***32.18** (*Parallel Eight Queens*) Revise Listing 22.11, EightQueens.java, to develop a parallel algorithm that finds all solutions for the Eight Queens problem. (*Hint*: Launch eight subtasks, each of which places the queen in a different column in the first row.)

Comprehensive

*****32.19** (*Sorting animation*) Write an animation for selection sort, insertion sort, and bubble sort, as shown in Figure 32.31. Create an array of integers `1`, `2`, . . . , `50`. Shuffle it randomly. Create a pane to display the array in a histogram. You should invoke each sort method in a separate thread. Each algorithm uses two nested loops. When the algorithm completes an iteration in the outer loop, put the thread to sleep for 0.5 seconds and redisplay the array in the histogram. Color the last bar in the sorted subarray.

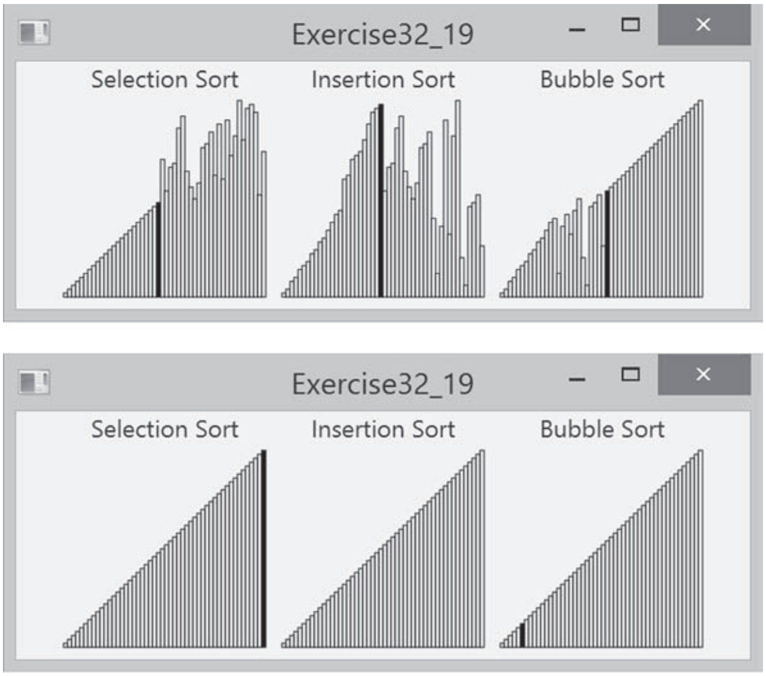
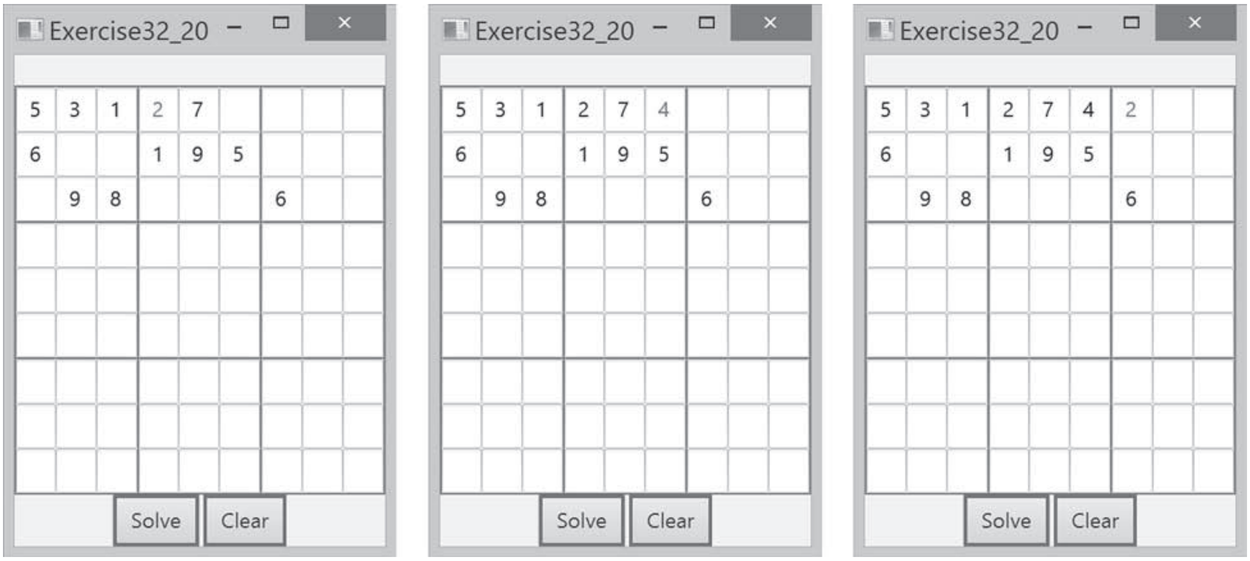

Exercise32_19
Selection Sort
Insertion Sort
Bubble Sort
Exercise32_19
Selection Sort
Insertion Sort
Bubble Sort

FIGURE 32.31 Three sorting algorithms are illustrated in the animation.

*****32.20** (*Sudoku search animation*) Modify Programming Exercise 22.21 to display the intermediate results of the search. Figure 32.32 gives a snapshot of an animation in progress with number **2** placed in the cell in Figure 32.32a, number **3** placed in the cell in Figure 32.32b, and number **3** placed in the cell in Figure 32.32c. The animation displays all the search steps.


Exercise32_20
Solve
Clear
Exercise32_20
Solve
Clear
Exercise32_20
Solve
Clear

5	3	1	2	7				
6			1	9	5			
	9	8				6		

(a)

5	3	1	2	7	4			
6			1	9	5			
	9	8				6		

(b)

5	3	1	2	7	4	2		
6			1	9	5			
	9	8				6		

(c)

FIGURE 32.32 The intermediate search steps are displayed in the animation for the Sudoku problem.

32.21 (*Combine colliding bouncing balls*) Rewrite Programming Exercise 20.5 using a thread to animate bouncing ball movements.

*****32.22** (*Eight Queens animation*) Modify Listing 22.11, EightQueens.java, to display the intermediate results of the search. As shown in Figure 32.33, the current row being searched is highlighted. Every one second, a new state of the chess board is displayed.

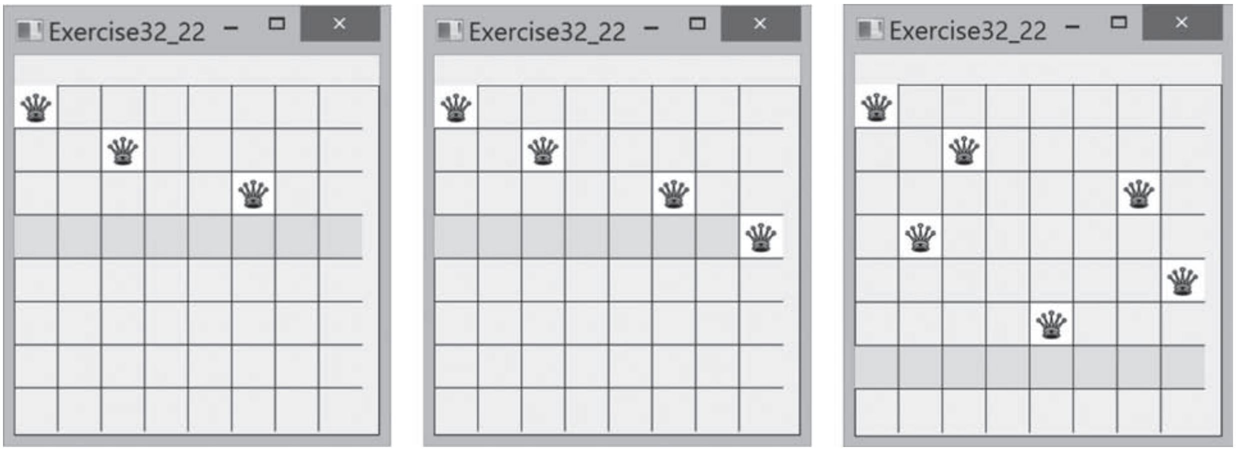


FIGURE 32.33 The intermediate search steps are displayed in the animation for the Eight Queens problem.

CHAPTER 33

NETWORKING

Objectives

- To explain the terms: TCP, IP, domain name, domain name server, stream-based communications, and packet-based communications (§33.2).
- To create servers using server sockets (§33.2.1) and clients using client sockets (§33.2.2).
- To implement Java networking programs using stream sockets (§33.2.3).
- To develop an example of a client/server application (§33.2.4).
- To obtain Internet addresses using the `InetAddress` class (§33.3).
- To develop servers for multiple clients (§33.4).
- To send and receive objects on a network (§33.5).
- To develop an interactive tic-tac-toe game played on the Internet (§33.6).

33.1 Introduction

Key Point

Computer networking is used to send and receive messages among computers on the Internet.

To browse the Web or send an email, your computer must be connected to the Internet. The *Internet* is the global network of millions of computers. Your computer can connect to the Internet through an Internet Service Provider (ISP) using a dialup, DSL, or cable modem, or through a local area network (LAN).

IP address

domain name

domain name server

When a computer needs to communicate with another computer, it needs to know the other computer's address. An *Internet Protocol* (IP) address uniquely identifies the computer on the Internet. An IP address consists of four dotted decimal numbers between `0` and `255`, such as `130.254.204.33.` Since it is not easy to remember so many numbers, they are often mapped to meaningful names called *domain names*, such as liang.armstrong.edu. Special servers called *Domain Name Servers* (DNS) on the Internet translate host names into IP addresses. When a computer contacts liang.armstrong.edu, it first asks the DNS to translate this domain name into a numeric IP address then sends the request using the IP address.

TCP

UDP

The Internet Protocol is a low-level protocol for delivering data from one computer to another across the Internet in packets. Two higher-level protocols used in conjunction with the IP are the *Transmission Control Protocol* (TCP) and the *User Datagram Protocol* (UDP). TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent. UDP is a standard, low-overhead, connectionless, host-to-host protocol that is used over the IP. UDP allows an application program on one computer to send a datagram to an application program on another computer.

stream-based communication

packet-based communication

Java supports both stream-based and packet-based communications. *Stream-based communications* use TCP for data transmission, whereas *packet-based communications* use UDP. Since TCP can detect lost transmissions and resubmit them, transmissions are lossless and reliable. UDP, in contrast, cannot guarantee lossless transmission. Stream-based communications are used in most areas of Java programming and are the focus of this chapter. Packet-based communications are introduced in Supplement III.P, Networking Using Datagram Protocol.

33.2 Client/Server Computing

Key Point

Java provides the **`ServerSocket`** *class for creating a server socket, and the* **`Socket`** *class for creating a client socket. Two programs on the Internet communicate through a server socket and a client socket using I/O streams.*

socket

Networking is tightly integrated in Java. The Java API provides the classes for creating sockets to facilitate program communications over the Internet. *Sockets* are the endpoints of logical connections between two hosts and can be used to send and receive data. Java treats socket communications much as it treats I/O operations; thus, programs can read from or write to sockets as easily as they can read from or write to files.

Network programming usually involves a server and one or more clients. The client sends requests to the server, and the server responds. The client begins by attempting to establish a connection to the server. The server can accept or deny the connection. Once a connection is established, the client and the server communicate through sockets.

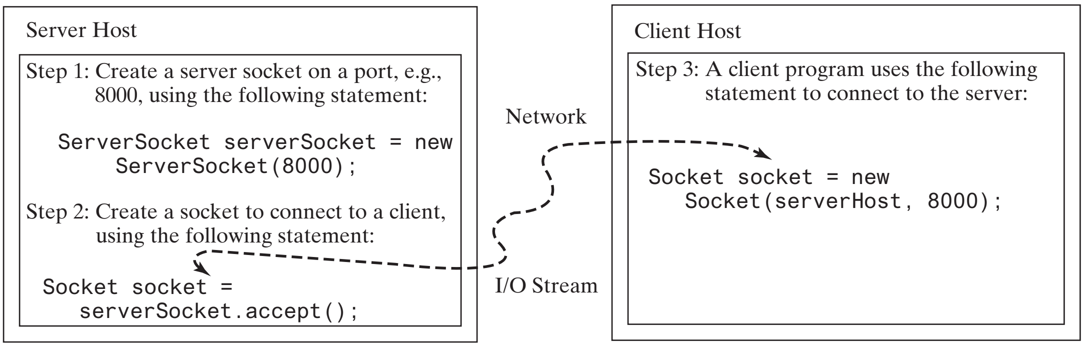
The server must be running when a client attempts to connect to the server. The server waits for a connection request from the client. The statements needed to create sockets on a server and on a client are shown in Figure 33.1.

33.2.1 Server Sockets

server socket

port

To establish a server, you need to create a *server socket* and attach it to a *port*, which is where the server listens for connections. The port identifies the TCP service on the socket. Port numbers range from 0 to 65536, but port numbers 0 to 1024 are reserved for privileged services.



Step 1: Create a server socket on a port, e.g., 8000, using the following statement:

```
ServerSocket serverSocket = new
        ServerSocket(8000);
```

Step 2: Create a socket to connect to a client, using the following statement:

```
Socket socket =
     serverSocket.accept();
```

Step 3: A client program uses the following statement to connect to the server:

```
Socket socket = new
     Socket(serverHost, 8000);
```

FIGURE 33.1 The server creates a server socket and, once a connection to a client is established, connects to the client with a client socket.

For instance, the email server runs on port 25, and the Web server usually runs on port 80. You can choose any port number that is not currently used by other programs. The following statement creates a server socket `serverSocket`:

```
ServerSocket serverSocket = new ServerSocket(port);
```

Note

Attempting to create a server socket on a port already in use would cause a `java.net.BindException`.

`BindException`

33.2.2 Client Sockets

After a server socket is created, the server can use the following statement to listen for connections:

```
Socket socket = serverSocket.accept();
```

This statement waits until a client connects to the server socket. The client issues the following statement to request a connection to a server:

connect to client

```
Socket socket = new Socket(serverName, port);
```

This statement opens a socket so that the client program can communicate with the server. `serverName` is the server's Internet host name or IP address. The following statement creates a socket on the client machine to connect to the host 130.254.204.33 at port 8000:

client socket

use IP address

```
Socket socket = new Socket("130.254.204.33", 8000);
```

Alternatively, you can use the domain name to create a socket, as follows:

use domain name

```
Socket socket = new Socket("liang.armstrong.edu", 8000);
```

When you create a socket with a host name, the JVM asks the DNS to translate the host name into the IP address.

Note

A program can use the host name `localhost` or the IP address `127.0.0.1` to refer to the machine on which a client is running.

`localhost`

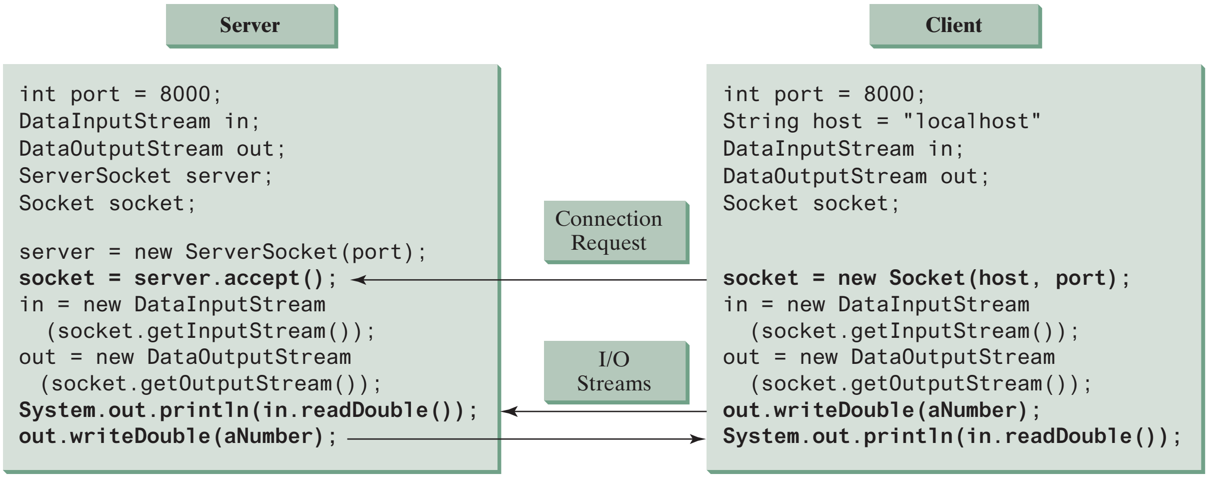
UnknownHostException

Note

The `Socket` constructor throws a `java.net.UnknownHostException` if the host cannot be found.

33.2.3 Data Transmission through Sockets

After the server accepts the connection, the communication between the server and the client is conducted in the same way as for I/O streams. The statements needed to create the streams and to exchange data between them are shown in Figure 33.2.



```
int port = 8000;
DataInputStream in;
DataOutputStream out;
ServerSocket server;
Socket socket;

server = new ServerSocket(port);
socket = server.accept();
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutputStream
  (socket.getOutputStream());
System.out.println(in.readDouble());
out.writeDouble(aNumber);
```

```
int port = 8000;
String host = "localhost"
DataInputStream in;
DataOutputStream out;
Socket socket;

socket = new Socket(host, port);
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutputStream
  (socket.getOutputStream());
out.writeDouble(aNumber);
System.out.println(in.readDouble());
```

FIGURE 33.2 The server and client exchange data through I/O streams on top of the socket.

To get an input stream and an output stream, use the `getInputStream()` and `getOutputStream()` methods on a socket object. For example, the following statements create an `InputStream` stream called `input` and an `OutputStream` stream called `output` from a socket:

```
InputStream input = socket.getInputStream();
OutputStream output = socket.getOutputStream();
```

The `InputStream` and `OutputStream` streams are used to read or write bytes. You can use `DataInputStream`, `DataOutputStream`, `BufferedReader`, and `PrintWriter` to wrap on the `InputStream` and `OutputStream` to read or write data, such as `int`, `double`, or `String`. The following statements, for instance, create the `DataInputStream` stream `input` and the `DataOutputstream output` to read and write primitive data values:

```
DataInputStream input = new DataInputStream
  (socket.getInputStream());
DataOutputStream output = new DataOutputStream
  (socket.getOutputStream());
```

The server can use `input.readDouble()` to receive a `double` value from the client, and `output.writeDouble(d)` to send the `double` value `d` to the client.

Tip

Recall that binary I/O is more efficient than text I/O because text I/O requires encoding and decoding. Therefore, it is better to use binary I/O for transmitting data between a server and a client to improve performance.

33.2.4 A Client/Server Example

This example presents a client program and a server program. The client sends data to a server. The server receives the data, uses it to produce a result, and then sends the result back to the client. The client displays the result on the console. In this example, the data sent from the client comprise the radius of a circle, and the result produced by the server is the area of the circle (see Figure 33.3).

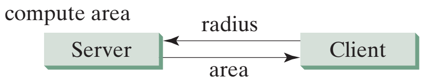

compute area
radius
Server
Client
area

FIGURE 33.3 The client sends the radius to the server; the server computes the area and sends it to the client.

The client sends the radius through a `DataOutputStream` on the output stream socket, and the server receives the radius through the `DataInputStream` on the input stream socket, as shown in Figure 33.4a. The server computes the area and sends it to the client through a `DataOutputStream` on the output stream socket, and the client receives the area through a `DataInputStream` on the input stream socket, as shown in Figure 33.4b. The server and client programs are given in Listings 33.1 and 33.2. Figure 33.5 contains a sample run of the server and the client.

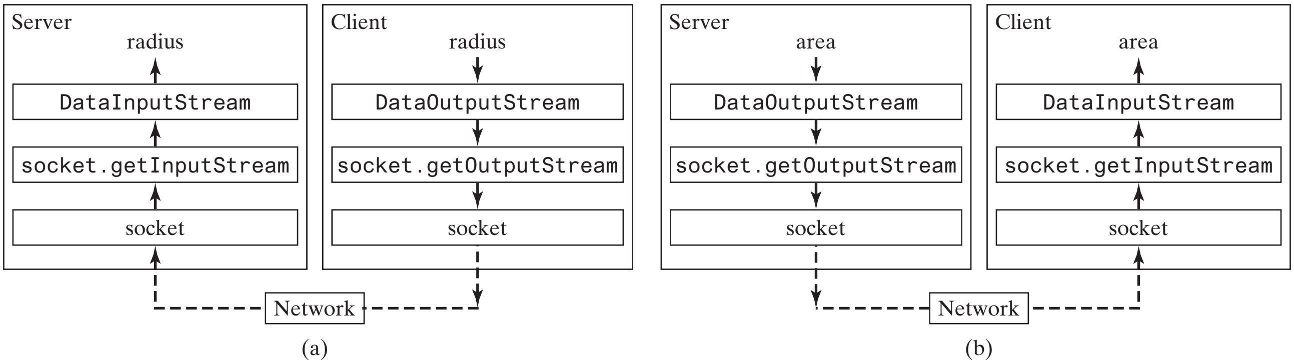
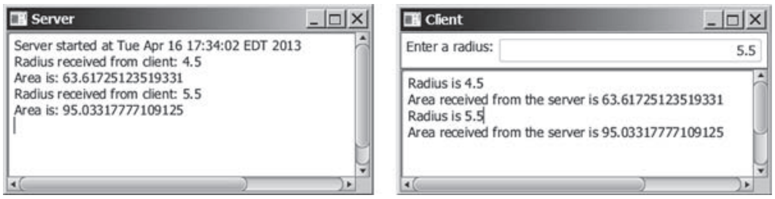

Server
radius
DataInputStream
socket.getInputStream
socket
Client
radius
DataOutputStream
socket.getOutputStream
socket
Network
(a)
Server
area
DataOutputStream
socket.getOutputStream
socket
Client
area
DataInputStream
socket.getInputStream
socket
Network
(b)

FIGURE 33.4 (a) The client sends the radius to the server. (b) The server sends the area to the client.


Server
Client
Enter a radius:

```
Server started at Tue Apr 16 17:34:02 EDT 2013
Radius received from client: 4.5
Area is: 63.61725123519331
Radius received from client: 5.5
Area is: 95.03317777109125
```

```
5.5
```

```
Radius is 4.5
Area received from the server is 63.61725123519331
Radius is 5.5
Area received from the server is 95.03317777109125
```

FIGURE 33.5 The client sends the radius to the server. The server receives it, computes the area, and sends the area to the client.

LISTING 33.1 Server.java

```java
import java.io.*;
import java.net.*;
import java.util.Date;
```

create server UI

server socket

update UI

connect client

input from client

output to client

read radius

write area

update UI

```java
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

public class Server extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Text area for displaying contents
    TextArea ta = new TextArea();

    // Create a scene and place it in the stage
    Scene scene = new Scene(new ScrollPane(ta), 450, 200);
    primaryStage.setTitle("Server"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread(() -> {
      try {
        // Create a server socket
        ServerSocket serverSocket = new ServerSocket(8000);
        Platform.runLater(() ->
          ta.appendText("Server started at " + new Date() + '\n'));

        // Listen for a connection request
        Socket socket = serverSocket.accept();

        // Create data input and output streams
        DataInputStream inputFromClient = new DataInputStream(
          socket.getInputStream());
        DataOutputStream outputToClient = new DataOutputStream(
          socket.getOutputStream());

        while (true) {
          // Receive radius from the client
          double radius = inputFromClient.readDouble();

          // Compute area
          double area = radius * radius * Math.PI;

          // Send area back to the client
          outputToClient.writeDouble(area);

          Platform.runLater(() -> {
            ta.appendText("Radius received from client: "
              + radius + '\n');
            ta.appendText("Area is: " + area + '\n');
          });
        }
      }
    catch(IOException ex) {
      ex.printStackTrace();
    }
  }).start();
 }
}
```

LISTING 33.2 `Client.java`

```
import java.io.*;
import java.net.*;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Client extends Application {
  // IO streams
  DataOutputStream toServer = null;
  DataInputStream fromServer = null;

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Panel p to hold the label and text field
    BorderPane paneForTextField = new BorderPane();
    paneForTextField.setPadding(new Insets(5, 5, 5, 5));
    paneForTextField.setStyle("-fx-border-color: green");
    paneForTextField.setLeft(new Label("Enter a radius: "));

    TextField tf = new TextField();
    tf.setAlignment(Pos.BOTTOM_RIGHT);
    paneForTextField.setCenter(tf);

    BorderPane mainPane = new BorderPane();
    // Text area to display contents
    TextArea ta = new TextArea();
    mainPane.setCenter(new ScrollPane(ta));
    mainPane.setTop(paneForTextField);

    // Create a scene and place it in the stage
    Scene scene = new Scene(mainPane, 450, 200);
    primaryStage.setTitle("Client"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    tf.setOnAction(e -> {
      try {
        // Get the radius from the text field
        double radius = Double.parseDouble(tf.getText().trim());

        // Send the radius to the server
        toServer.writeDouble(radius);
        toServer.flush();

        // Get area from the server
        double area = fromServer.readDouble();

        // Display to the text area
        ta.appendText("Radius is " + radius + "\n");
        ta.appendText("Area received from the server is "
          + area + '\n');
```

create UI

handle action event

read radius

write radius

read area

request connection

input from server

output to server

```
59       }
60       catch (IOException ex) {
61         System.err.println(ex);
62       }
63     });
64 
65     try {
66       // Create a socket to connect to the server
67       Socket socket = new Socket("localhost", 8000);
68       // Socket socket = new Socket("130.254.204.36", 8000);
69       // Socket socket = new Socket("drake.Armstrong.edu", 8000);
70 
71       // Create an input stream to receive data from the server
72       fromServer = new DataInputStream(socket.getInputStream());
73 
74       // Create an output stream to send data to the server
75       toServer = new DataOutputStream(socket.getOutputStream());
76     }
77     catch (IOException ex) {
78       ta.appendText(ex.toString() + '\n');
79     }
80   }
81 }
```

You start the server program first then start the client program. In the client program, enter a radius in the text field and press *Enter* to send the radius to the server. The server computes the area and sends it back to the client. This process is repeated until one of the two programs terminates.

The networking classes are in the package `java.net`. You should import this package when writing Java network programs.

The `Server` class creates a `ServerSocket serverSocket` and attaches it to port 8000 using this statement (line 26 in Server.java):

```
ServerSocket serverSocket = new ServerSocket(8000);
```

The server then starts to listen for connection requests, using the following statement (line 31 in Server.java):

```
Socket socket = serverSocket.accept();
```

The server waits until the client requests a connection. After it is connected, the server reads the radius from the client through an input stream, computes the area, and sends the result to the client through an output stream. The `ServerSocket accept()` method takes time to execute. It is not appropriate to run this method in the JavaFX application thread. So, we place it in a separate thread (lines 23–59). The statements for updating GUI needs to run from the JavaFX application thread using the `Platform.runLater` method (lines 27–28, 49–53).

The `Client` class uses the following statement to create a socket that will request a connection to the server on the same machine (localhost) at port 8000 (line 67 in Client.java).

```
Socket socket = new Socket("localhost", 8000);
```

If you run the server and the client on different machines, replace `localhost` with the server machine's host name or IP address. In this example, the server and the client are running on the same machine.

If the server is not running, the client program terminates with a `java.net.ConnectException`. After it is connected, the client gets input and output streams—wrapped by data input and output streams—in order to receive and send data to the server.

If you receive a `java.net.BindException` when you start the server, the server port is currently in use. You need to terminate the process that is using the server port then restart the server.

Note

When you create a server socket, you have to specify a port (e.g., 8000) for the socket. When a client connects to the server (line 67 in Client.java), a socket is created on the client. This socket has its own local port. This port number (e.g., 2047) is automatically chosen by the JVM, as shown in Figure 33.6.

client socket port

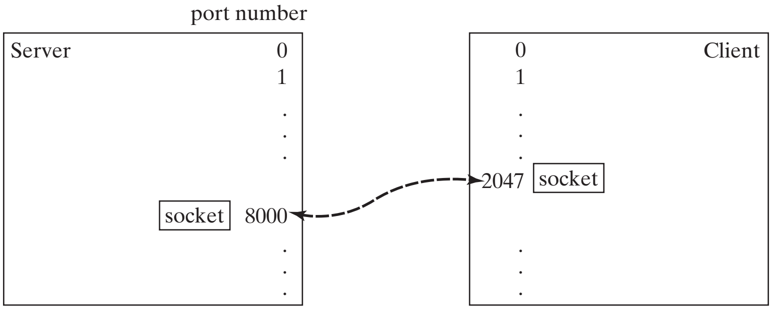


FIGURE 33.6 The JVM automatically chooses an available port to create a socket for the client.

To see the local port on the client, insert the following statement in line 70 in Client.java.

```
System.out.println("local port: " + socket.getLocalPort());
```

Check Point

33.2.1 How do you create a server socket? What port numbers can be used? What happens if a requested port number is already in use? Can a port connect to multiple clients?

33.2.2 What are the differences between a server socket and a client socket?

33.2.3 How does a client program initiate a connection?

33.2.4 How does a server accept a connection?

33.2.5 How are data transferred between a client and a server?

33.3 The InetAddress Class

Key Point

The server program can use the **`InetAddress`** *class to obtain the information about the IP address and host name for the client.*

Occasionally, you would like to know who is connecting to the server. You can use the `InetAddress` class to find the client's host name and IP address. The `InetAddress` class models an IP address. You can use the following statement in the server program to get an instance of `InetAddress` on a socket that connects to the client:

```
InetAddress inetAddress = socket.getInetAddress();
```

Next, you can display the client's host name and IP address, as follows:

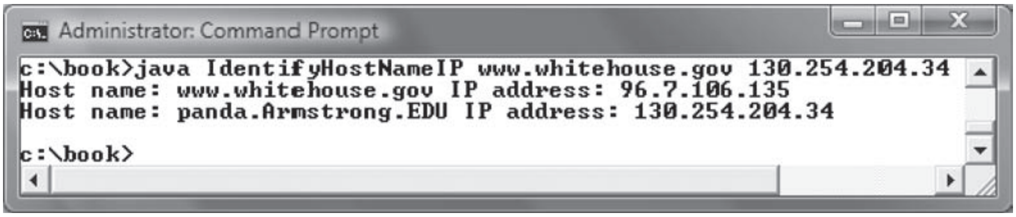
```
System.out.println("Client's host name is " +
  inetAddress.getHostName());
```

```
System.out.println("Client's IP Address is " +
  inetAddress.getHostAddress());
```

You can also create an instance of `InetAddress` from a host name or IP address using the static `getByName` method. For example, the following statement creates an `InetAddress` for the host `liang.armstrong.edu`.

```
InetAddress address = InetAddress.getByName("liang.armstrong.edu");
```

Listing 33.3 gives a program that identifies the host name and IP address of the arguments you pass in from the command line. Line 7 creates an `InetAddress` using the `getByName` method. Lines 8 and 9 use the `getHostName` and `getHostAddress` methods to get the host's name and IP address. Figure 33.7 shows a sample run of the program.



```
c:\book>java IdentifyHostNameIP www.whitehouse.gov 130.254.204.34
Host name: www.whitehouse.gov IP address: 96.7.106.135
Host name: panda.Armstrong.EDU IP address: 130.254.204.34

c:\book>
```

Figure 33.7 The program identifies host names and IP addresses.

Listing 33.3 IdentifyHostNameIP.java

get an InetAddress
get host name
get host IP

```
 1  import java.net.*;
 2
 3  public class IdentifyHostNameIP {
 4    public static void main(String[] args) {
 5      for (int i = 0; i < args.length; i++) {
 6        try {
 7          InetAddress address = InetAddress.getByName(args[i]);
 8          System.out.print("Host name: " + address.getHostName() + " ");
 9          System.out.println("IP address: " + address.getHostAddress());
10        }
11        catch (UnknownHostException ex) {
12          System.err.println("Unknown host or IP address " + args[i]);
13        }
14      }
15    }
16  }
```

Check Point

33.3.1 How do you obtain an instance of `InetAddress`?

33.3.2 What methods can you use to get the IP address and hostname from an `InetAddress`?

33.4 Serving Multiple Clients

Key Point

A server can serve multiple clients. The connection to each client is handled by one thread.

Multiple clients are quite often connected to a single server at the same time. Typically, a server runs continuously on a server computer, and clients from all over the Internet can connect to it. You can use threads to handle the server's multiple clients simultaneously—simply create a thread for each connection. Here is how the server handles the establishment of a connection:

```
while (true) {
  Socket socket = serverSocket.accept(); // Connect to a client
  Thread thread = new ThreadClass(socket);
  thread.start();
}
```

The server socket can have many connections. Each iteration of the `while` loop creates a new connection. Whenever a connection is established, a new thread is created to handle communication between the server and the new client, and this allows multiple connections to run at the same time.

Listing 33.4 creates a server class that serves multiple clients simultaneously. For each connection, the server starts a new thread. This thread continuously receives input (the radius of a circle) from clients and sends the results (the area of the circle) back to them (see Figure 33.8). The client program is the same as in Listing 33.2. A sample run of the server with two clients is shown in Figure 33.9.

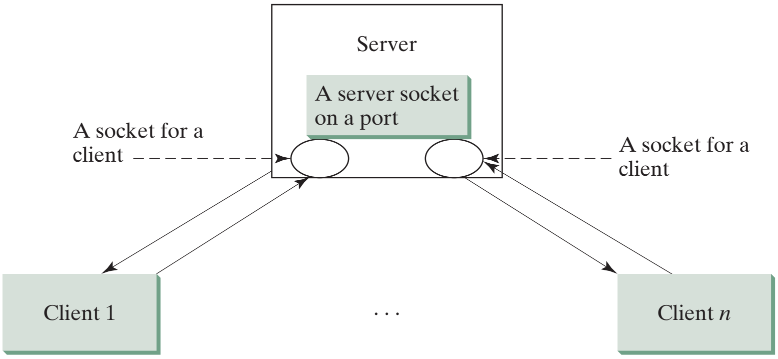
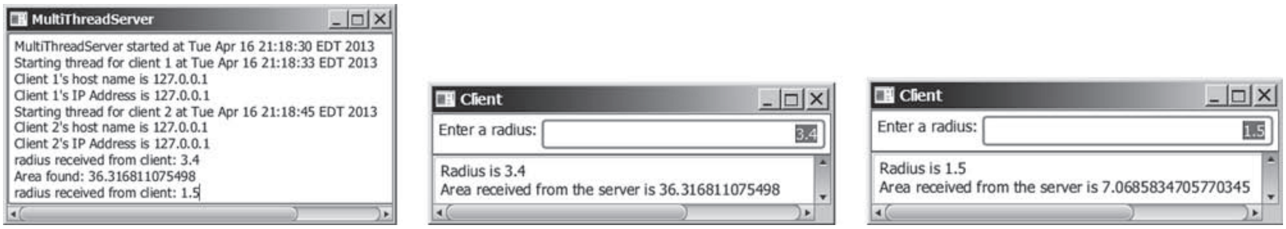


FIGURE 33.8 Multithreading enables a server to handle multiple independent clients.



```
MultiThreadServer started at Tue Apr 16 21:18:30 EDT 2013
Starting thread for client 1 at Tue Apr 16 21:18:33 EDT 2013
Client 1's host name is 127.0.0.1
Client 1's IP Address is 127.0.0.1
Starting thread for client 2 at Tue Apr 16 21:18:45 EDT 2013
Client 2's host name is 127.0.0.1
Client 2's IP Address is 127.0.0.1
radius received from client: 3.4
Area found: 36.316811075498
radius received from client: 1.5
```

```
Radius is 3.4
Area received from the server is 36.316811075498
```

```
Radius is 1.5
Area received from the server is 7.0685834705770345
```

FIGURE 33.9 The server spawns a thread in order to serve a client.

LISTING 33.4 MultiThreadServer.java

```
import java.io.*;
import java.net.*;
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

```

```java
public class MultiThreadServer extends Application {
  // Text area for displaying contents
  private TextArea ta = new TextArea();

  // Number a client
  private int clientNo = 0;

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place it in the stage
    Scene scene = new Scene(new ScrollPane(ta), 450, 200);
    primaryStage.setTitle("MultiThreadServer"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread( () -> {
      try {
        // Create a server socket
```

server socket

```java
        ServerSocket serverSocket = new ServerSocket(8000);
        ta.appendText("MultiThreadServer started at "
          + new Date() + '\n');

        while (true) {
          // Listen for a new connection request
```

connect client

```java
          Socket socket = serverSocket.accept();

          // Increment clientNo
          clientNo++;

```

update GUI

```java
          Platform.runLater( () -> {
            // Display the client number
            ta.appendText("Starting thread for client " + clientNo +
              " at " + new Date() + '\n');

            // Find the client's host name, and IP address
```

network information

```java
            InetAddress inetAddress = socket.getInetAddress();
            ta.appendText("Client " + clientNo + "'s host name is "
              + inetAddress.getHostName() + "\n");
            ta.appendText("Client " + clientNo + "'s IP Address is"
              + inetAddress.getHostAddress() + "\n");
          });

          // Create and start a new thread for the connection
```

create task

```java
          new Thread(new HandleAClient(socket)).start();
        }
      }
      catch(IOException ex) {
        System.err.println(ex);
      }
```

start thread

```java
    }).start();
  }

  // Define the thread class for handling new connection
  class HandleAClient implements Runnable {
```

task class

```java
    private Socket socket; // A connected socket

    /** Construct a thread */
    public HandleAClient(Socket socket) {
      this.socket = socket;
    }

```

```
 72      /** Run a thread */
 73      public void run() {
 74        try {
 75          // Create data input and output streams
 76          DataInputStream inputFromClient = new DataInputStream(
 77            socket.getInputStream());
 78          DataOutputStream outputToClient = new DataOutputStream(
 79            socket.getOutputStream());
 80
 81          // Continuously serve the client
 82          while (true) {
 83            // Receive radius from the client
 84            double radius = inputFromClient.readDouble();
 85
 86            // Compute area
 87            double area = radius * radius * Math.PI;
 88
 89            // Send area back to the client
 90            outputToClient.writeDouble(area);
 91
 92            Platform.runLater(() -> {
 93              ta.appendText("radius received from client: " +
 94                radius + '\n');
 95              ta.appendText("Area found: " + area + '\n');
 96            });
 97          }
 98        }
 99        catch(IOException ex) {
100          ex.printStackTrace();
101        }
102      }
103    }
104  }
```

I/O

update GUI

The server creates a server socket at port 8000 (line 29) and waits for a connection (line 35). When a connection with a client is established, the server creates a new thread to handle the communication (line 54). It then waits for another connection in an infinite `while` loop (lines 33–55).

The threads, which run independently of one another, communicate with designated clients. Each thread creates data input and output streams that receive and send data to a client.

33.4.1 How do you make a server serve multiple clients?

Check Point

33.5 Sending and Receiving Objects

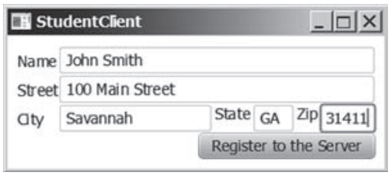
A program can send and receive objects from another program.

Key Point

In the preceding examples, you learned how to send and receive data of primitive types. You can also send and receive objects using `ObjectOutputStream` and `ObjectInputStream` on socket streams. To enable passing, the objects must be serializable. The following example demonstrates how to send and receive objects.

The example consists of three classes: StudentAddress.java (Listing 33.5), StudentClient.java (Listing 33.6), and StudentServer.java (Listing 33.7). The client program collects student information from the client and sends it to a server, as shown in Figure 33.10.

The `StudentAddress` class contains the student information: name, street, city, state, and zip. The `StudentAddress` class implements the `Serializable` interface. Therefore, a `StudentAddress` object can be sent and received using the object output and input streams.



Name John Smith
Street 100 Main Street
City Savannah State GA Zip 31411

FIGURE 33.10 The client sends the student information in an object to the server.

LISTING 33.5 StudentAddress.java

serialized

```
public class StudentAddress implements java.io.Serializable {
  private String name;
  private String street;
  private String city;
  private String state;
  private String zip;

  public StudentAddress(String name, String street, String city,
    String state, String zip) {
    this.name = name;
    this.street = street;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getName() {
    return name;
  }

  public String getStreet() {
    return street;
  }

  public String getCity() {
    return city;
  }

  public String getState() {
    return state;
  }

  public String getZip() {
    return zip;
  }
}
```

The client sends a `StudentAddress` object through an `ObjectOutputStream` on the output stream socket, and the server receives the `Student` object through the `ObjectInputStream` on the input stream socket, as shown in Figure 33.11. The client uses the `writeObject` method in the `ObjectOutputStream` class to send data about a student to the server, and the server receives the student's information using the `readObject` method in the `ObjectInputStream` class. The server and client programs are given in Listings 33.6 and 33.7.

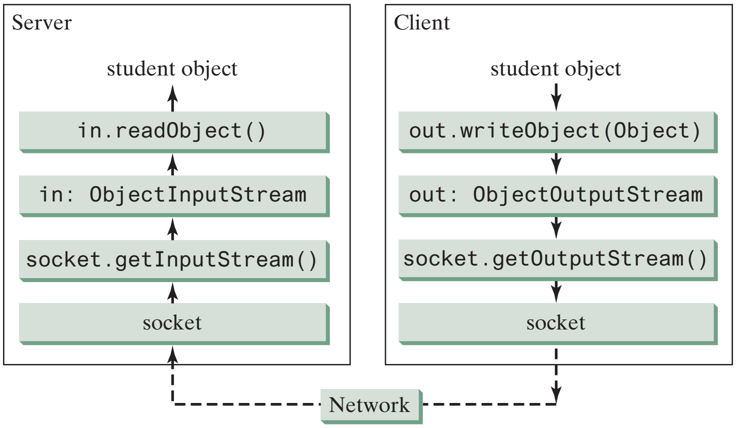

Server
student object
in.readObject()
in: ObjectInputStream
socket.getInputStream()
socket
Client
student object
out.writeObject(Object)
out: ObjectOutputStream
socket.getOutputStream()
socket
Network

FIGURE 33.11 The client sends a `StudentAddress` object to the server.

LISTING 33.6 StudentClient.java

```java
import java.io.*;
import java.net.*;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.HPos;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class StudentClient extends Application {
  private TextField tfName = new TextField();
  private TextField tfStreet = new TextField();
  private TextField tfCity = new TextField();
  private TextField tfState = new TextField();
  private TextField tfZip = new TextField();

  // Button for sending a student to the server
  private Button btRegister = new Button("Register to the Server");

  // Host name or ip
  String host = "localhost";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    GridPane pane = new GridPane();
    pane.add(new Label("Name"), 0, 0);
    pane.add(tfName, 1, 0);
    pane.add(new Label("Street"), 0, 1);
    pane.add(tfStreet, 1, 1);
    pane.add(new Label("City"), 0, 2);

```

create UI

register listener

server socket

output stream

send to server

```
    HBox hBox = new HBox(2);
    pane.add(hBox, 1, 2);
    hBox.getChildren().addAll(tfCity, new Label("State"), tfState,
      new Label("Zip"), tfZip);
    pane.add(btRegister, 1, 3);
    GridPane.setHalignment(btRegister, HPos.RIGHT);

    pane.setAlignment(Pos.CENTER);
    tfName.setPrefColumnCount(15);
    tfStreet.setPrefColumnCount(15);
    tfCity.setPrefColumnCount(10);
    tfState.setPrefColumnCount(2);
    tfZip.setPrefColumnCount(3);

    btRegister.setOnAction(new ButtonListener());

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 450, 200);
    primaryStage.setTitle("StudentClient"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /** Handle button action */
  private class ButtonListener implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e) {
      try {
        // Establish connection with the server
        Socket socket = new Socket(host, 8000);

        // Create an output stream to the server
        ObjectOutputStream toServer =
          new ObjectOutputStream(socket.getOutputStream());

        // Get text field
        String name = tfName.getText().trim();
        String street = tfStreet.getText().trim();
        String city = tfCity.getText().trim();
        String state = tfState.getText().trim();
        String zip = tfZip.getText().trim();

        // Create a Student object and send to the server
        StudentAddress s =
          new StudentAddress(name, street, city, state, zip);
        toServer.writeObject(s);
      }
      catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Listing 33.7 StudentServer.java

```
import java.io.*;
import java.net.*;

public class StudentServer {
 5    private ObjectOutputStream outputToFile;
 6    private ObjectInputStream inputFromClient;
 7
 8    public static void main(String[] args) {
 9      new StudentServer();
10    }
11
12    public StudentServer() {
13      try {
14        // Create a server socket
15        ServerSocket serverSocket = new ServerSocket(8000);
16        System.out.println("Server started ");
17
18        // Create an object output stream
19        outputToFile = new ObjectOutputStream(
20          new FileOutputStream("student.dat", true));
21
22        while (true) {
23          // Listen for a new connection request
24          Socket socket = serverSocket.accept();
25
26          // Create an input stream from the socket
27          inputFromClient =
28            new ObjectInputStream(socket.getInputStream());
29
30          // Read from input
31          Object object = inputFromClient.readObject();
32
33          // Write to the file
34          outputToFile.writeObject(object);
35          System.out.println("A new student object is stored");
36        }
37      }
38      catch(ClassNotFoundException ex) {
39        ex.printStackTrace();
40      }
41      catch(IOException ex) {
42        ex.printStackTrace();
43      }
44      finally {
45        try {
46          inputFromClient.close();
47          outputToFile.close();
48        }
49        catch (Exception ex) {
50          ex.printStackTrace();
51        }
52      }
53    }
54  }
```

server socket

output to file

connect to client

input stream

get from client

write to file

On the client side, when the user clicks the *Register to the Server* button, the client creates a socket to connect to the host (line 67), creates an `ObjectOutputStream` on the output stream of the socket (lines 70 and 71), and invokes the `writeObject` method to send the `StudentAddress` object to the server through the object output stream (line 83).

On the server side, when a client connects to the server, the server creates an `ObjectInputStream` on the input stream of the socket (lines 27 and 28), invokes the `readObject` method to receive the `StudentAddress` object through the object input stream (line 31), and writes the object to a file (line 34).

Check Point

33.5.1 How does a server receive connection from a client? How does a client connect to a server?

33.5.2 How do you find the host name of a client program from the server?

33.5.3 How do you send and receive an object?

33.6 Case Study: Distributed Tic-Tac-Toe Games

Key Point

This section develops a program that enables two players to play the tic-tac-toe game on the Internet.

In Section 16.12, Case Study: Developing a Tic-Tac-Toe Game, you developed a program for a tic-tac-toe game that enables two players to play the game on the same machine. In this section, you will learn how to develop a distributed tic-tac-toe game using multithreads and networking with socket streams. A distributed tic-tac-toe game enables users to play on different machines from anywhere on the Internet.

You need to develop a server for multiple clients. The server creates a server socket and accepts connections from every two players to form a session. Each session is a thread that communicates with the two players and determines the status of the game. The server can establish any number of sessions, as shown in Figure 33.12.

For each session, the first client connecting to the server is identified as player 1 with token `X`, and the second client connecting is identified as player 2 with token `O`. The server notifies the players of their respective tokens. Once two clients are connected to it, the server starts a thread to facilitate the game between the two players by performing the steps repeatedly, as shown in Figure 33.13.

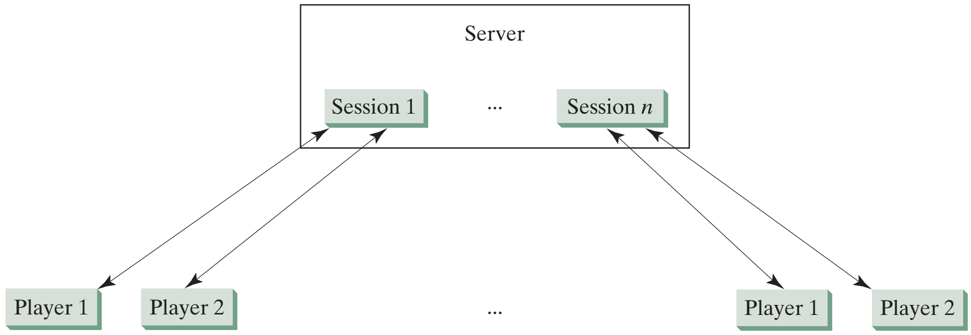


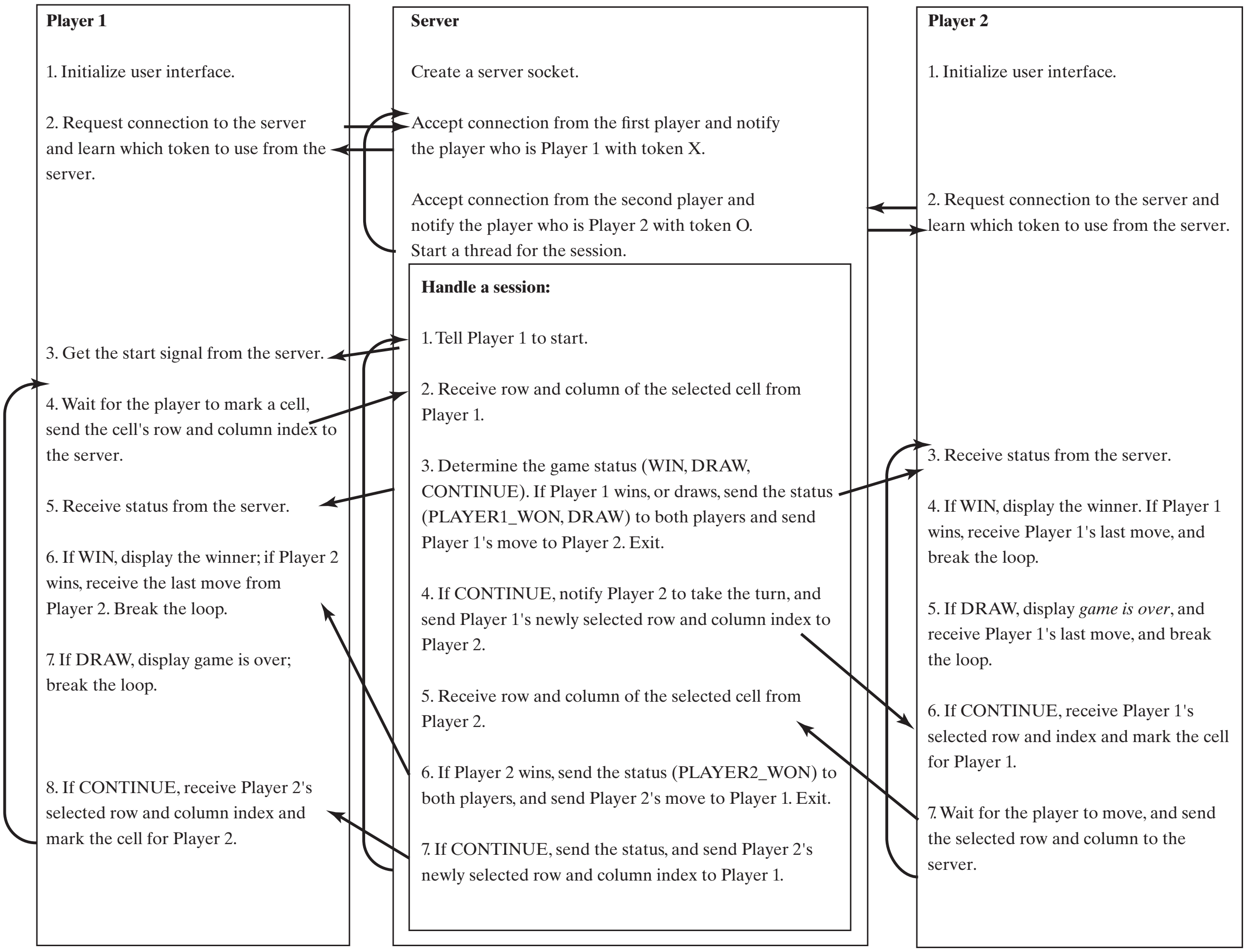
FIGURE 33.12 The server can create many sessions, each of which facilitates a tic-tac-toe game for two players.

The server does not have to be a graphical component, but creating it in a GUI in which game information can be viewed is user friendly. You can create a scroll pane to hold a text area in the GUI and display game information in the text area. The server creates a thread to handle a game session when two players are connected to the server.

The client is responsible for interacting with the players. It creates a user interface with nine cells and displays the game title and status to the players in the labels. The client class is very similar to the `TicTacToe` class presented in the case study in Listing 16.13. However, the client in this example does not determine the game status (win or draw); it simply passes the moves to the server and receives the game status from the server.

Based on the foregoing analysis, you can create the following classes:

- `TicTacToeServer` serves all the clients in Listing 33.9.
- `HandleASession` facilitates the game for two players. This class is defined in Listing 33.9, TicTacToeServer.java.



Player 1

1. Initialize user interface.

2. Request connection to the server and learn which token to use from the server.

3. Get the start signal from the server.

4. Wait for the player to mark a cell, send the cell's row and column index to the server.

5. Receive status from the server.

6. If WIN, display the winner; if Player 2 wins, receive the last move from Player 2. Break the loop.

7. If DRAW, display game is over; break the loop.

8. If CONTINUE, receive Player 2's selected row and column index and mark the cell for Player 2.

Server

Create a server socket.

Accept connection from the first player and notify the player who is Player 1 with token X.

Accept connection from the second player and notify the player who is Player 2 with token O. Start a thread for the session.

Handle a session:

1. Tell Player 1 to start.

2. Receive row and column of the selected cell from Player 1.

3. Determine the game status (WIN, DRAW, CONTINUE). If Player 1 wins, or draws, send the status (PLAYER1_WON, DRAW) to both players and send Player 1's move to Player 2. Exit.

4. If CONTINUE, notify Player 2 to take the turn, and send Player 1's newly selected row and column index to Player 2.

5. Receive row and column of the selected cell from Player 2.

6. If Player 2 wins, send the status (PLAYER2_WON) to both players, and send Player 2's move to Player 1. Exit.

7. If CONTINUE, send the status, and send Player 2's newly selected row and column index to Player 1.

Player 2

1. Initialize user interface.

2. Request connection to the server and learn which token to use from the server.

3. Receive status from the server.

4. If WIN, display the winner. If Player 1 wins, receive Player 1's last move, and break the loop.

5. If DRAW, display *game is over*, and receive Player 1's last move, and break the loop.

6. If CONTINUE, receive Player 1's selected row and index and mark the cell for Player 1.

7. Wait for the player to move, and send the selected row and column to the server.

FIGURE 33.13 The server starts a thread to facilitate communications between the two players.

- `TicTacToeClient` models a player in Listing 33.10.
- `Cell` models a cell in the game. It is an inner class in `TicTacToeClient`.
- `TicTacToeConstants` is an interface that defines the constants shared by all the classes in the example in Listing 33.8.

The relationships of these classes are shown in Figure 33.14.

LISTING 33.8 TicTacToeConstants.java

```java
public interface TicTacToeConstants {
  public static int PLAYER1 = 1; // Indicate player 1
  public static int PLAYER2 = 2; // Indicate player 2
  public static int PLAYER1_WON = 1; // Indicate player 1 won
  public static int PLAYER2_WON = 2; // Indicate player 2 won
  public static int DRAW = 3; // Indicate a draw
  public static int CONTINUE = 4; // Indicate to continue
}
```

LISTING 33.9 TicTacToeServer.java

```java
import java.io.*;
import java.net.*;
```

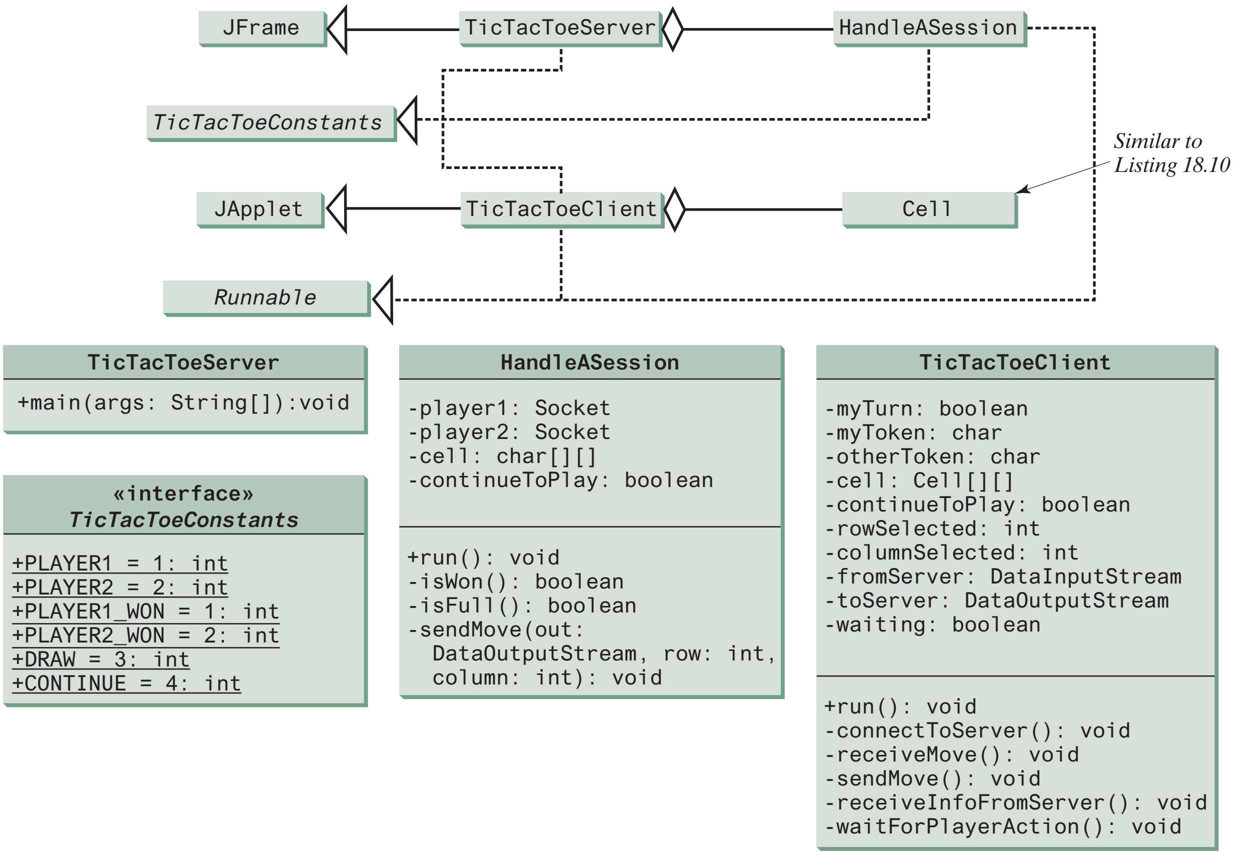

JFrame
TicTacToeServer
HandleASession
TicTacToeConstants
Similar to Listing 18.10
JApplet
TicTacToeClient
Cell
Runnable
TicTacToeServer
+main(args: String[]):void
«interface»
TicTacToeConstants
+PLAYER1 = 1: int
+PLAYER2 = 2: int
+PLAYER1_WON = 1: int
+PLAYER2_WON = 2: int
+DRAW = 3: int
+CONTINUE = 4: int
HandleASession
-player1: Socket
-player2: Socket
-cell: char[][]
-continueToPlay: boolean
+run(): void
-isWon(): boolean
-isFull(): boolean
-sendMove(out: DataOutputStream, row: int, column: int): void
TicTacToeClient
-myTurn: boolean
-myToken: char
-otherToken: char
-cell: Cell[][]
-continueToPlay: boolean
-rowSelected: int
-columnSelected: int
-fromServer: DataInputStream
-toServer: DataOutputStream
-waiting: boolean
+run(): void
-connectToServer(): void
-receiveMove(): void
-sendMove(): void
-receiveInfoFromServer(): void
-waitForPlayerAction(): void

FIGURE 33.14 `TicTacToeServer` creates an instance of `HandleASession` for each session of two players. `TicTacToeClient` creates nine cells in the UI.

create UI

```java
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

public class TicTacToeServer extends Application
    implements TicTacToeConstants {
  private int sessionNo = 1; // Number a session

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    TextArea taLog = new TextArea();

    // Create a scene and place it in the stage
    Scene scene = new Scene(new ScrollPane(taLog), 450, 200);
    primaryStage.setTitle("TicTacToeServer"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread( () -> {
      try {
        // Create a server socket
      ServerSocket serverSocket = new ServerSocket(8000);
      Platform.runLater(() -> taLog.appendText(new Date() +
        ": Server started at socket 8000\n"));

      // Ready to create a session for every two players
      while (true) {
        Platform.runLater(() -> taLog.appendText(new Date() +
         ": Wait for players to join session " + sessionNo + '\n'));

        // Connect to player 1
        Socket player1 = serverSocket.accept();

        Platform.runLater(() -> {
          taLog.appendText(new Date() + ": Player 1 joined session "
            + sessionNo + '\n');
          taLog.appendText("Player 1's IP address" +
            player1.getInetAddress().getHostAddress() + '\n');
        });

        // Notify that the player is Player 1
        new DataOutputStream(
          player1.getOutputStream()).writeInt(PLAYER1);

        // Connect to player 2
        Socket player2 = serverSocket.accept();

        Platform.runLater(() -> {
          taLog.appendText(new Date() +
            ": Player 2 joined session " + sessionNo + '\n');
          taLog.appendText("Player 2's IP address" +
            player2.getInetAddress().getHostAddress() + '\n');
        });

        // Notify that the player is Player 2
        new DataOutputStream(
          player2.getOutputStream()).writeInt(PLAYER2);

        // Display this session and increment session number
        Platform.runLater(() ->
          taLog.appendText(new Date() +
            ": Start a thread for session " + sessionNo++ + '\n'));

        // Launch a new thread for this session of two players
        new Thread(new HandleASession(player1, player2)).start();
      }
    }
    catch(IOException ex) {
      ex.printStackTrace();
    }
  }).start();
}

// Define the thread class for handling a new session for two players
class HandleASession implements Runnable, TicTacToeConstants {
  private Socket player1;
  private Socket player2;

  // Create and initialize cells
  private char[][] cell = new char[3][3];

  private DataInputStream fromPlayer1;
```

server socket

connect to client

to player1

connect to client

to player2

a session for two players

```
    private DataOutputStream toPlayer1;
    private DataInputStream fromPlayer2;
    private DataOutputStream toPlayer2;

    // Continue to play
    private boolean continueToPlay = true;

    /** Construct a thread */
    public HandleASession(Socket player1, Socket player2) {
      this.player1 = player1;
      this.player2 = player2;

      // Initialize cells
      for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
          cell[i][j] = ' ';
    }

    /** Implement the run() method for the thread */
    public void run() {
      try {
        // Create data input and output streams
```

IO streams

```
        DataInputStream fromPlayer1 = new DataInputStream(
          player1.getInputStream());
        DataOutputStream toPlayer1 = new DataOutputStream(
          player1.getOutputStream());
        DataInputStream fromPlayer2 = new DataInputStream(
          player2.getInputStream());
        DataOutputStream toPlayer2 = new DataOutputStream(
          player2.getOutputStream());

        // Write anything to notify player 1 to start
        // This is just to let player 1 know to start
        toPlayer1.writeInt(1);

        // Continuously serve the players and determine and report
        // the game status to the players
        while (true) {
          // Receive a move from player 1
          int row = fromPlayer1.readInt();
          int column = fromPlayer1.readInt();
          cell[row][column] = 'X';

          // Check if Player 1 wins
```

X won?

```
          if (isWon('X')) {
            toPlayer1.writeInt(PLAYER1_WON);
            toPlayer2.writeInt(PLAYER1_WON);
            sendMove(toPlayer2, row, column);
            break; // Break the loop
          }
```

Is full?

```
          else if (isFull()) { // Check if all cells are filled
            toPlayer1.writeInt(DRAW);
            toPlayer2.writeInt(DRAW);
            sendMove(toPlayer2, row, column);
            break;
          }
          else {
            // Notify player 2 to take the turn
            toPlayer2.writeInt(CONTINUE);

          // Send player 1's selected row and column to player 2
          sendMove(toPlayer2, row, column);
        }

        // Receive a move from Player 2
        row = fromPlayer2.readInt();
        column = fromPlayer2.readInt();
        cell[row][column] = 'O';

        // Check if Player 2 wins
        if (isWon('O')) {
```

O won?

```
          toPlayer1.writeInt(PLAYER2_WON);
          toPlayer2.writeInt(PLAYER2_WON);
          sendMove(toPlayer1, row, column);
          break;
        }
        else {
          // Notify player 1 to take the turn
          toPlayer1.writeInt(CONTINUE);

          // Send player 2's selected row and column to player 1
          sendMove(toPlayer1, row, column);
        }
      }
    }
    catch(IOException ex) {
      ex.printStackTrace();
    }
  }

  /** Send the move to other player */
  private void sendMove(DataOutputStream out, int row, int column)
```

send a move

```
      throws IOException {
    out.writeInt(row); // Send row index
    out.writeInt(column); // Send column index
  }

  /** Determine if the cells are all occupied */
  private boolean isFull() {
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        if (cell[i][j] == ' ')
          return false; // At least one cell is not filled

    // All cells are filled
    return true;
  }

  /** Determine if the player with the specified token wins */
  private boolean isWon(char token) {
    // Check all rows
    for (int i = 0; i < 3; i++)
      if ((cell[i][0] == token)
          && (cell[i][1] == token)
          && (cell[i][2] == token)) {
        return true;
      }

    /** Check all columns */
    for (int j = 0; j < 3; j++)
```

```
          if ((cell[0][j] == token)
              && (cell[1][j] == token)
              && (cell[2][j] == token)) {
            return true;
          }

        /** Check major diagonal */
        if ((cell[0][0] == token)
            && (cell[1][1] == token)
            && (cell[2][2] == token)) {
          return true;
        }

        /** Check subdiagonal */
        if ((cell[0][2] == token)
            && (cell[1][1] == token)
            && (cell[2][0] == token)) {
          return true;
        }

        /** All checked, but no winner */
        return false;
      }
    }
}
```

LISTING 33.10 TicTacToeClient.java

```
import java.io.*;
import java.net.*;
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
import javafx.scene.shape.Line;
import javafx.stage.Stage;

public class TicTacToeClient extends Application
    implements TicTacToeConstants {
  // Indicate whether the player has the turn
  private boolean myTurn = false;

  // Indicate the token for the player
  private char myToken = ' ';

  // Indicate the token for the other player
  private char otherToken = ' ';

  // Create and initialize cells
  private Cell[][] cell = new Cell[3][3];

  // Create and initialize a title label
  private Label lblTitle = new Label();

  // Create and initialize a status label
  private Label lblStatus = new Label();

  // Indicate selected row and column by the current move
  private int rowSelected;
  private int columnSelected;

  // Input and output streams from/to server
  private DataInputStream fromServer;
  private DataOutputStream toServer;

  // Continue to play?
  private boolean continueToPlay = true;

  // Wait for the player to mark a cell
  private boolean waiting = true;

  // Host name or ip
  private String host = "localhost";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Pane to hold cell
    GridPane pane = new GridPane();
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        pane.add(cell[i][j] = new Cell(i, j), j, i);

    BorderPane borderPane = new BorderPane();
    borderPane.setTop(lblTitle);
    borderPane.setCenter(pane);
    borderPane.setBottom(lblStatus);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 320, 350);
    primaryStage.setTitle("TicTacToeClient"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    // Connect to the server
    connectToServer();
  }

  private void connectToServer() {
    try {
      // Create a socket to connect to the server
      Socket socket = new Socket(host, 8000);

      // Create an input stream to receive data from the server
      fromServer = new DataInputStream(socket.getInputStream());

      // Create an output stream to send data to the server
      toServer = new DataOutputStream(socket.getOutputStream());
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
```

create UI

connect to server

input from server

output to server

```java

    // Control the game on a separate thread
    new Thread(() -> {
      try {
        // Get notification from the server
        int player = fromServer.readInt();

        // Am I player 1 or 2?
        if (player == PLAYER1) {
          myToken = 'X';
          otherToken = 'O';
          Platform.runLater(() -> {
            lblTitle.setText("Player 1 with token 'X'");
            lblStatus.setText("Waiting for player 2 to join");
          });

          // Receive startup notification from the server
          fromServer.readInt(); // Whatever read is ignored

          // The other player has joined
          Platform.runLater(() ->
            lblStatus.setText("Player 2 has joined. I start first"));

          // It is my turn
          myTurn = true;
        }
        else if (player == PLAYER2) {
          myToken = 'O';
          otherToken = 'X';
          Platform.runLater(() -> {
            lblTitle.setText("Player 2 with token 'O'");
            lblStatus.setText("Waiting for player 1 to move");
          });
        }

        // Continue to play
        while (continueToPlay) {
          if (player == PLAYER1) {
            waitForPlayerAction(); // Wait for player 1 to move
            sendMove(); // Send the move to the server
            receiveInfoFromServer(); // Receive info from the server
          }
          else if (player == PLAYER2) {
            receiveInfoFromServer(); // Receive info from the server
            waitForPlayerAction(); // Wait for player 2 to move
            sendMove(); // Send player 2's move to the server
          }
        }
      }
      catch (Exception ex) {
        ex.printStackTrace();
      }
    }).start();
  }

  /** Wait for the player to mark a cell */
  private void waitForPlayerAction() throws InterruptedException {
    while (waiting) {
      Thread.sleep(100);
    }

    waiting = true;
  }

  /** Send this player's move to the server */
  private void sendMove() throws IOException {
    toServer.writeInt(rowSelected); // Send the selected row
    toServer.writeInt(columnSelected); // Send the selected column
  }

  /** Receive info from the server */
  private void receiveInfoFromServer() throws IOException {
    // Receive game status
    int status = fromServer.readInt();

    if (status == PLAYER1_WON) {
      // Player 1 won, stop playing
      continueToPlay = false;
      if (myToken == 'X') {
        Platform.runLater(() -> lblStatus.setText("I won! (X)"));
      }
      else if (myToken == 'O') {
        Platform.runLater(() ->
          lblStatus.setText("Player 1 (X) has won!"));
        receiveMove();
      }
    }
    else if (status == PLAYER2_WON) {
      // Player 2 won, stop playing
      continueToPlay = false;
      if (myToken == 'O') {
        Platform.runLater(() -> lblStatus.setText("I won! (O)"));
      }
      else if (myToken == 'X') {
        Platform.runLater(() ->
          lblStatus.setText("Player 2 (O) has won!"));
        receiveMove();
      }
    }
    else if (status == DRAW) {
      // No winner, game is over
      continueToPlay = false;
      Platform.runLater(() ->
        lblStatus.setText("Game is over, no winner!"));

      if (myToken == 'O') {
        receiveMove();
      }
    }
    else {
      receiveMove();
      Platform.runLater(() -> lblStatus.setText("My turn"));
      myTurn = true; // It is my turn
    }
  }

  private void receiveMove() throws IOException {
    // Get the other player's move
    int row = fromServer.readInt();
    int column = fromServer.readInt();
```

model a cell

register listener

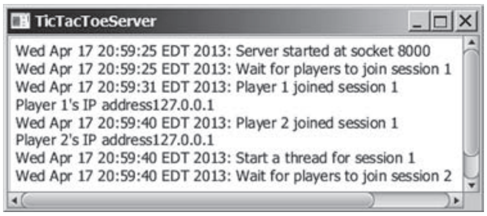
draw X

draw O

```
      Platform.runLater(() -> cell[row][column].setToken(otherToken));
    }

    // An inner class for a cell
    public class Cell extends Pane {
      // Indicate the row and column of this cell in the board
      private int row;
      private int column;

      // Token used for this cell
      private char token = ' ';

      public Cell(int row, int column) {
        this.row = row;
        this.column = column;
        this.setPrefSize(2000, 2000); // What happens without this?
        setStyle("-fx-border-color: black"); // Set cell's border
        this.setOnMouseClicked(e -> handleMouseClick());
      }

      /** Return token */
      public char getToken() {
        return token;
      }

      /** Set a new token */
      public void setToken(char c) {
        token = c;
        repaint();
      }

      protected void repaint() {
        if (token == 'X') {
          Line line1 = new Line(10, 10,
            this.getWidth() - 10, this.getHeight() - 10);
          line1.endXProperty().bind(this.widthProperty().subtract(10));
          line1.endYProperty().bind(this.heightProperty().subtract(10));
          Line line2 = new Line(10, this.getHeight() - 10,
            this.getWidth() - 10, 10);
          line2.startYProperty().bind(
            this.heightProperty().subtract(10));
          line2.endXProperty().bind(this.widthProperty().subtract(10));

          // Add the lines to the pane
          this.getChildren().addAll(line1, line2);
        }
        else if (token == 'O') {
          Ellipse ellipse = new Ellipse(this.getWidth() / 2,
            this.getHeight() / 2, this.getWidth() /  2 - 10,
            this.getHeight() / 2 - 10);
          ellipse.centerXProperty().bind(
            this.widthProperty().divide(2));
          ellipse.centerYProperty().bind(
            this.heightProperty().divide(2));
          ellipse.radiusXProperty().bind(
            this.widthProperty().divide(2).subtract(10));
          ellipse.radiusYProperty().bind(
            this.heightProperty().divide(2).subtract(10));
          ellipse.setStroke(Color.BLACK);
          ellipse.setFill(Color.WHITE);

        getChildren().add(ellipse); // Add the ellipse to the pane
      }
    }

    /* Handle a mouse click event */
    private void handleMouseClick() {
      // If cell is not occupied and the player has the turn
      if (token == ' ' && myTurn) {
        setToken(myToken); // Set the player's token in the cell
        myTurn = false;
        rowSelected = row;
        columnSelected = column;
        lblStatus.setText("Waiting for the other player to move");
        waiting = false; // Just completed a successful move
      }
    }
  }
}
```

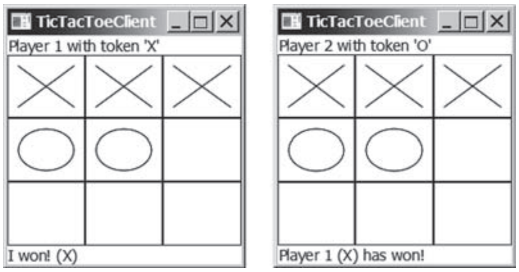
mouse clicked handler

The server can serve any number of sessions simultaneously. Each session takes care of two players. The client can be deployed to run as a Java applet. To run a client as a Java applet from a Web browser, the server must run from a Web server. Figures 33.15 and 33.16 show sample runs of the server and the clients.



```
Wed Apr 17 20:59:25 EDT 2013: Server started at socket 8000
Wed Apr 17 20:59:25 EDT 2013: Wait for players to join session 1
Wed Apr 17 20:59:31 EDT 2013: Player 1 joined session 1
Player 1's IP address127.0.0.1
Wed Apr 17 20:59:40 EDT 2013: Player 2 joined session 1
Player 2's IP address127.0.0.1
Wed Apr 17 20:59:40 EDT 2013: Start a thread for session 1
Wed Apr 17 20:59:40 EDT 2013: Wait for players to join session 2
```

FIGURE 33.15 `TicTacToeServer` accepts connection requests and creates sessions to serve pairs of players.



Player 1 with token 'X'

I won! (X)

Player 2 with token 'O'

Player 1 (X) has won!

FIGURE 33.16 `TicTacToeClient` can run as an applet or standalone.

The `TicTacToeConstants` interface defines the constants shared by all the classes in the project. Each class that uses the constants needs to implement the interface. Centrally defining constants in an interface is a common practice in Java.

Once a session is established, the server receives moves from the players in alternation. Upon receiving a move from a player, the server determines the status of the game. If the game is not finished, the server sends the status (`CONTINUE`) and the player's move to the other player. If the game is won or a draw, the server sends the status (`PLAYER1_WON`, `PLAYER2_WON`, or `DRAW`) to both players.

The implementation of Java network programs at the socket level is tightly synchronized. An operation to send data from one machine requires an operation to receive data from the other machine. As shown in this example, the server and the client are tightly synchronized to send or receive data.

Check Point

33.6.1 What would happen if the preferred size for a cell is not set in line 227 in Listing 33.10?

33.6.2 If a player does not have the turn but clicks on an empty cell, what will the client program in Listing 33.10 do?

KEY TERMS

client socket	33-3
domain name	33-2
domain name server	33-2
localhost	33-3
IP address	33-2
port	33-2
packet-based communication	33-2
server socket	33-2
socket	33-2
stream-based communication	33-2
TCP	33-2
UDP	33-2

CHAPTER SUMMARY

1. Java supports stream sockets and datagram sockets. *Stream sockets* use TCP (Transmission Control Protocol) for data transmission, whereas *datagram sockets* use UDP (User Datagram Protocol). Since TCP can detect lost transmissions and resubmit them, transmissions are lossless and reliable. UDP, in contrast, cannot guarantee lossless transmission.

2. To create a server, you must first obtain a server socket, using `new ServerSocket(port)`. After a server socket is created, the server can start to listen for connections, using the `accept()` method on the server socket. The client requests a connection to a server by using `new Socket(serverName, port)` to create a client socket.

3. Stream socket communication is very much like input/output stream communication after the connection between a server and a client is established. You can obtain an input stream using the `getInputStream()` method and an output stream using the `getOutputStream()` method on the socket.

4. A server must often work with multiple clients at the same time. You can use threads to handle the server's multiple clients simultaneously by creating a thread for each connection.

QUIZ

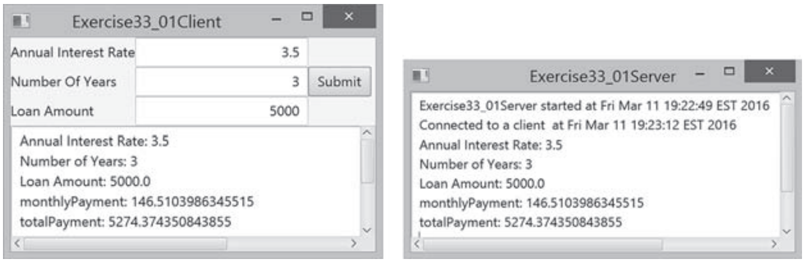
Answer the quiz for this chapter online at book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

Section 33.2

***33.1** (*Loan server*) Write a server for a client. The client sends loan information (annual interest rate, number of years, and loan amount) to the server (see Figure 33.17a). The server computes monthly payment and total payment, and sends them back to the client (see Figure 33.17b). Name the client Exercise33_01Client and the server Exercise33_01Server.



Annual Interest Rate 3.5
Number Of Years 3
Loan Amount 5000

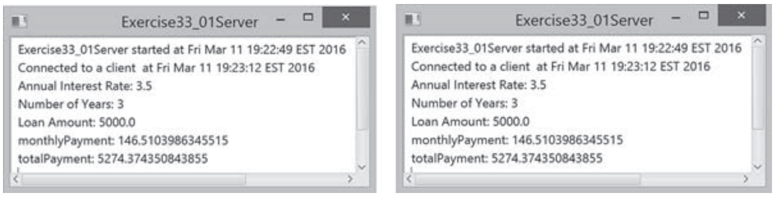
```
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

```
Exercise33_01Server started at Fri Mar 11 19:22:49 EST 2016
Connected to a client  at Fri Mar 11 19:23:12 EST 2016
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

(a) (b)

FIGURE 33.17 The client in (a) sends the annual interest rate, number of years, and loan amount to the server and receives the monthly payment and total payment from the server in (b).

***33.2** (*BMI server*) Write a server for a client. The client sends the weight and height for a person to the server (see Figure 33.18a). The server computes BMI (Body Mass Index) and sends back to the client a string that reports the BMI (see Figure 33.18b). See Section 3.8 for computing BMI. Name the client Exercise33_02Client and the server Exercise33_02Server.



```
Exercise33_01Server started at Fri Mar 11 19:22:49 EST 2016
Connected to a client  at Fri Mar 11 19:23:12 EST 2016
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

```
Exercise33_01Server started at Fri Mar 11 19:22:49 EST 2016
Connected to a client  at Fri Mar 11 19:23:12 EST 2016
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

(a) (b)

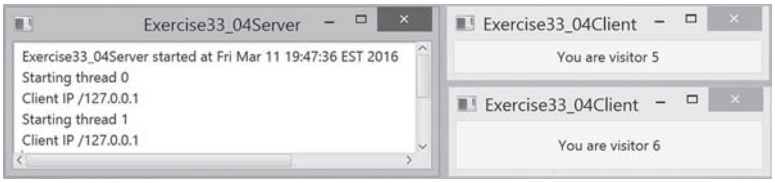
FIGURE 33.18 The client in (a) sends the weight and height of a person to the server and receives the BMI from the server in (b).

Sections 33.3 and 33.4

***33.3** (*Loan server for multiple clients*) Revise Programming Exercise 33.1 to write a server for multiple clients.

Section 33.5

33.4 (*Count clients*) Write a server that tracks the number of the clients connected to the server. When a new connection is established, the count is incremented by 1. The count is stored using a random-access file. Write a client program that receives the count from the server and displays a message, such as “You are visitor number 11”, as shown in Figure 33.19. Name the client Exercise33_04Client and the server Exercise33_04Server.



```
Exercise33_04Server started at Fri Mar 11 19:47:36 EST 2016
Starting thread 0
Client IP /127.0.0.1
Starting thread 1
Client IP /127.0.0.1
```

```
You are visitor 5
```

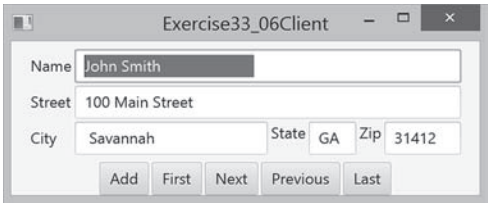
```
You are visitor 6
```

FIGURE 33.19 The client displays how many times the server has been accessed. The server stores the count.

33.5 (*Send loan information in an object*) Revise Exercise 33.1 for the client to send a loan object that contains annual interest rate, number of years, and loan amount and for the server to send the monthly payment and total payment.

Section 33.6

33.6 (*Display and add addresses*) Develop a client/server application to view and add addresses, as shown in Figure 33.20.



Name: John Smith
Street: 100 Main Street
City: Savannah
State: GA
Zip: 31412

FIGURE 33.20 You can view and add an address.

- Use the `StudentAddress` class defined in Listing 33.5 to hold the name, street, city, state, and zip in an object.
- The user can use the buttons *First*, *Next*, *Previous*, and *Last* to view an address, and the *Add* button to add a new address.
- Limit the concurrent connections to two clients.

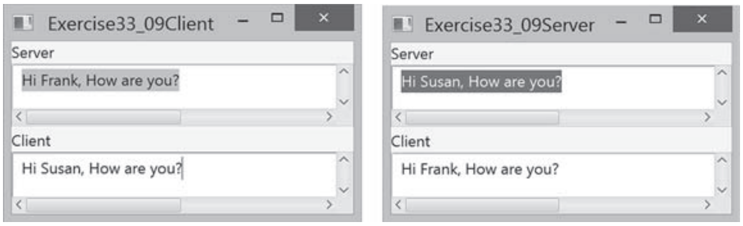
Name the client Exercise33_06Client and the server Exercise33_6Server.

***33.7** (*Transfer last 100 numbers in an array*) Programming Exercise 22.12 retrieves the last 100 prime numbers from a file **PrimeNumbers.dat**. Write a client program that requests the server to send the last 100 prime numbers in an array. Name the server program Exercise33_07Server and the client program Exercise33_07Client. Assume the numbers of the `long` type are stored in **PrimeNumbers.dat** in binary format.

***33.8** (*Transfer last 100 numbers in an `ArrayList`*) Programming Exercise 24.12 retrieves the last 100 prime numbers from a file **PrimeNumbers.dat**. Write a client program that requests the server to send the last 100 prime numbers in an `ArrayList`. Name the server program Exercise33_08Server and the client program Exercise33_08Client. Assume the numbers of the `long` type are stored in **PrimeNumbers.dat** in binary format.

Section 33.7

****33.9** (*Chat*) Write a program that enables two users to chat. Implement one user as the server (see Figure 33.21a) and the other as the client (see Figure 33.21b). The server has two text areas: one for entering text, and the other (noneditable) for displaying text received from the client. When the user presses the *Enter* key, the current line is sent to the client. The client has two text areas: one (noneditable) for displaying text from the server and the other for entering text. When the user presses the *Enter* key, the current line is sent to the server. Name the client Exercise33_09Client and the server Exercise33_09Server.



Exercise33_09Client:

Server: Hi Frank, How are you?

Client: Hi Susan, How are you?

Exercise33_09Server:

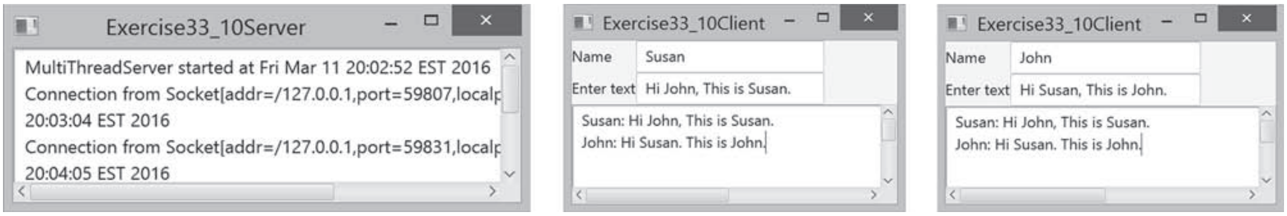
Server: Hi Susan, How are you?

Client: Hi Frank, How are you?

(a) (b)

FIGURE 33.21 The server and client send text to and receive text from each other.

*****33.10** (*Multiple client chat*) Write a program that enables any number of clients to chat. Implement one server that serves all the clients, as shown in Figure 33.22. Name the client Exercise33_10Client and the server Exercise33_10Server.



Exercise33_10Server:

```
MultiThreadServer started at Fri Mar 11 20:02:52 EST 2016
Connection from Socket[addr=/127.0.0.1,port=59807,localp
20:03:04 EST 2016
Connection from Socket[addr=/127.0.0.1,port=59831,localp
20:04:05 EST 2016
```

Exercise33_10Client:

Name: Susan

Enter text: Hi John, This is Susan.

```
Susan: Hi John, This is Susan.
John: Hi Susan. This is John.
```

Exercise33_10Client:

Name: John

Enter text: Hi Susan, This is John.

```
Susan: Hi John, This is Susan.
John: Hi Susan. This is John.
```

(a) (b) (c)

FIGURE 33.22 The server starts in (a) with three clients in (b) and (c).

CHAPTER

34

JAVA DATABASE PROGRAMMING

Objectives

- To understand the concepts of databases and database management systems (§34.2).
- To understand the relational data model: relational data structures, constraints, and languages (§34.2).
- To use SQL to create and drop tables and to retrieve and modify data (§34.3).
- To learn how to load a driver, connect to a database, execute statements, and process result sets using JDBC (§34.4).
- To use prepared statements to execute precompiled SQL statements (§34.5).
- To use callable statements to execute stored SQL procedures and functions (§34.6).
- To explore database metadata using the `DatabaseMetaData` and `ResultSetMetaData` interfaces (§34.7).

34.1 Introduction

Key Point

Java provides the API for developing database applications that works with any relational database systems.

You may have heard a lot about database systems. Database systems are everywhere. Your social security information is stored in a database by the government. If you shop online, your purchase information is stored in a database by the company. If you attend a university, your academic information is stored in a database by the university. Database systems not only store data, they also provide means of accessing, updating, manipulating, and analyzing data. Your social security information is updated periodically, and you can register for courses online. Database systems play an important role in society and in commerce.

This chapter introduces database systems, the SQL language, and how database applications can be developed using Java. If you already know SQL, you can skip Sections 34.2 and 34.3.

34.2 Relational Database Systems

Key Point

SQL is the standard database language for defining and accessing databases.

database system

A *database system* consists of a database, the software that stores and manages data in the database, and the application programs that present data and enable the user to interact with the database system, as shown in Figure 34.1.

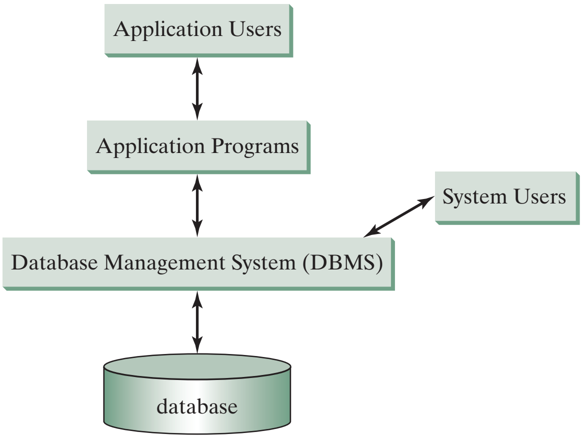


FIGURE 34.1 A database system consists of data, database management software, and application programs.

A *database* is a repository of data that form information. When you purchase a database system—such as MySQL, Oracle, IBM's DB2 and Informix, Microsoft SQL Server, or Sybase—from a software vendor, you actually purchase the software comprising a *database management system* (DBMS). Database management systems are designed for use by professional programmers and are not suitable for ordinary customers. Application programs are built on top of the DBMS for customers to access and update the database. Thus, application programs can be viewed as the *interfaces* between the database system and its users. Application programs may be stand-alone GUI applications or Web applications and may access several different database systems in the network, as shown in Figure 34.2.

DBMS

Most of today's database systems are *relational database systems*. They are based on the relational data model, which has three key components: structure, integrity, and language.

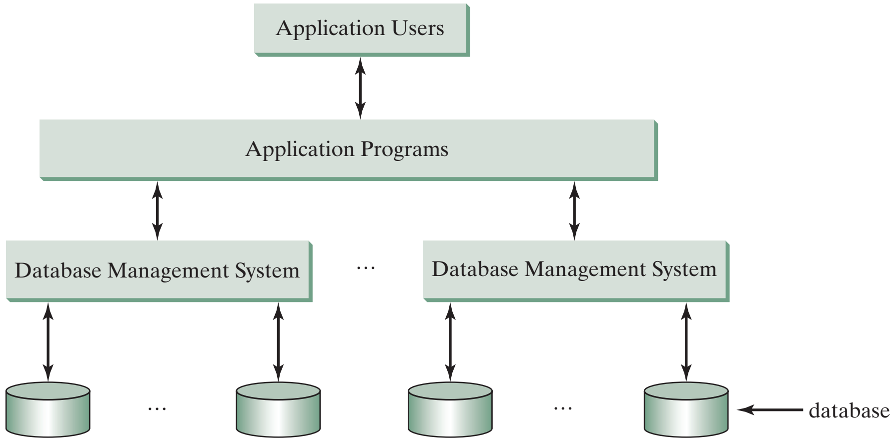


FIGURE 34.2 An application program can access multiple database systems.

Structure defines the representation of the data. *Integrity* imposes constraints on the data. *Language* provides the means for accessing and manipulating data.

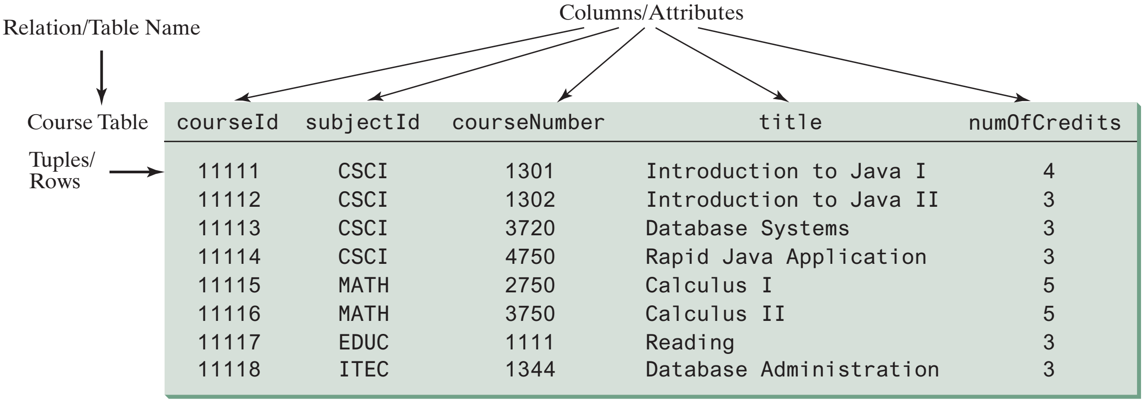
34.2.1 Relational Structures

relational model

The relational model is built around a simple and natural structure. A *relation* is actually a table that consists of nonduplicate rows. Tables are easy to understand and use. The relational model provides a simple yet powerful way to represent data.

tuple
attribute

A row of a table represents a *record*, and a column of a table represents the *value of a single attribute* of the record. In relational database theory, a row is called a *tuple*, and a column is called an *attribute*. Figure 34.3 shows a sample table that stores information about the courses offered by a university. The table has eight tuples, and each tuple has five attributes.


Relation/Table Name
Columns/Attributes
Course Table
Tuples/
Rows

courseId	subjectId	courseNumber	title	numOfCredits
11111	CSCI	1301	Introduction to Java I	4
11112	CSCI	1302	Introduction to Java II	3
11113	CSCI	3720	Database Systems	3
11114	CSCI	4750	Rapid Java Application	3
11115	MATH	2750	Calculus I	5
11116	MATH	3750	Calculus II	5
11117	EDUC	1111	Reading	3
11118	ITEC	1344	Database Administration	3

FIGURE 34.3 A table has a table name, column names, and rows.

Tables describe the relationship among data. Each row in a table represents a record of related data. For example, “11111,” “CSCI,” “1301,” “Introduction to Java I,” and “4” are related to form a record (the first row in Figure 34.3) in the `Course` table. Just as the data in the same row are related, so too data in different tables may be related through common attributes. Suppose the database has two other tables, `Student` and `Enrollment`, as shown in

Figures 34.4 and 34.5. The `Course` table and the `Enrollment` table are related through their common attribute `courseId`, and the `Enrollment` table and the `Student` table are related through `ssn`.

Student Table

ssn	firstName	mi	lastName	phone	birthDate	street	zipCode	deptID
444111110	Jacob	R	Smith	9129219434	1985-04-09	99 Kingston Street	31435	BIOL
444111111	John	K	Stevenson	9129219434	null	100 Main Street	31411	BIOL
444111112	George	K	Smith	9129213454	1974-10-10	1200 Abercorn St.	31419	CS
444111113	Frank	E	Jones	9125919434	1970-09-09	100 Main Street	31411	BIOL
444111114	Jean	K	Smith	9129219434	1970-02-09	100 Main Street	31411	CHEM
444111115	Josh	R	Woo	7075989434	1970-02-09	555 Franklin St.	31411	CHEM
444111116	Josh	R	Smith	9129219434	1973-02-09	100 Main Street	31411	BIOL
444111117	Joy	P	Kennedy	9129229434	1974-03-19	103 Bay Street	31412	CS
444111118	Toni	R	Peterson	9129229434	1964-04-29	103 Bay Street	31412	MATH
444111119	Patrick	R	Stoneman	9129229434	1969-04-29	101 Washington St.	31435	MATH
444111120	Rick	R	Carter	9125919434	1986-04-09	19 West Ford St.	31411	BIOL

Figure 34.4 A `Student` table stores student information.

`Enrollment` Table

ssn	courseId	dateRegistered	grade
444111110	11111	2004-03-19	A
444111110	11112	2004-03-19	B
444111110	11113	2004-03-19	C
444111111	11111	2004-03-19	D
444111111	11112	2004-03-19	F
444111111	11113	2004-03-19	A
444111112	11114	2004-03-19	B
444111112	11115	2004-03-19	C
444111112	11116	2004-03-19	D
444111113	11111	2004-03-19	A
444111113	11113	2004-03-19	A
444111114	11115	2004-03-19	B
444111115	11115	2004-03-19	F
444111115	11116	2004-03-19	F
444111116	11111	2004-03-19	D
444111117	11111	2004-03-19	D
444111118	11111	2004-03-19	A
444111118	11112	2004-03-19	D
444111118	11113	2004-03-19	B

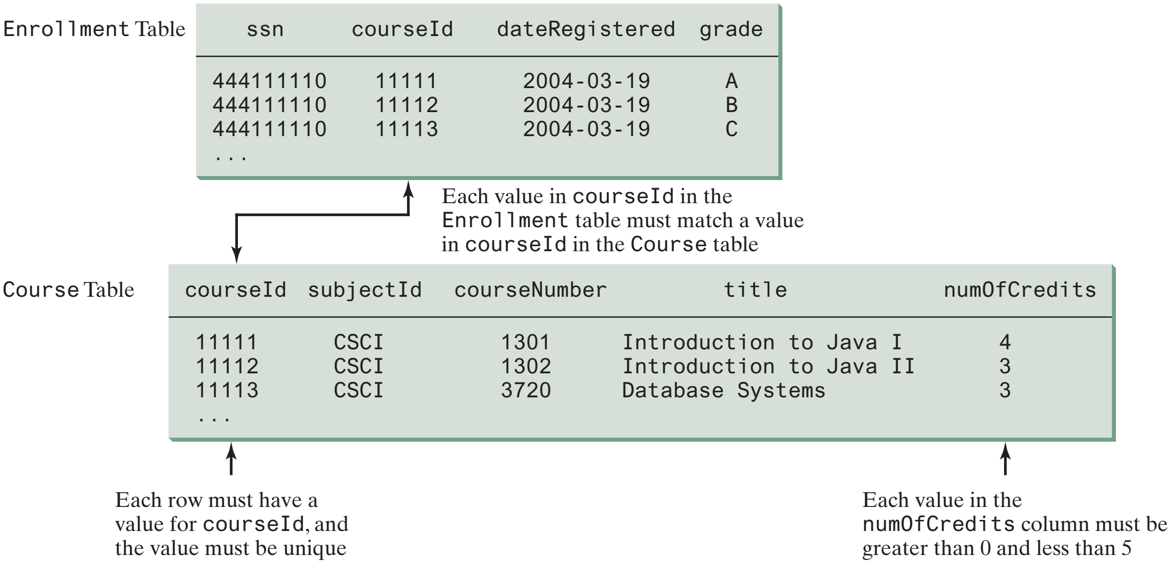
Figure 34.5 An `Enrollment` table stores student enrollment information.

34.2.2 Integrity Constraints

integrity constraint

An *integrity constraint* imposes a condition that all the legal values in a table must satisfy. Figure 34.6 shows an example of some integrity constraints in the `Subject` and `Course` tables.

In general, there are three types of constraints: domain constraints, primary key constraints, and foreign key constraints. *Domain constraints* and *primary key constraints* are known as *intrarelational constraints*, meaning that a constraint involves only one relation. The *foreign key constraint* is *interrelational*, meaning that a constraint involves more than one relation.



Enrollment Table

ssn	courseId	dateRegistered	grade
444111110	11111	2004-03-19	A
444111110	11112	2004-03-19	B
444111110	11113	2004-03-19	C
...			

Each value in `courseId` in the `Enrollment` table must match a value in `courseId` in the `Course` table

Course Table

courseId	subjectId	courseNumber	title	numOfCredits
11111	CSCI	1301	Introduction to Java I	4
11112	CSCI	1302	Introduction to Java II	3
11113	CSCI	3720	Database Systems	3
...				

Each row must have a value for `courseId`, and the value must be unique

Each value in the `numOfCredits` column must be greater than 0 and less than 5

FIGURE 34.6 The **`Enrollment`** table and the **`Course`** table have integrity constraints.

Domain Constraints

domain constraint

Domain constraints specify the permissible values for an attribute. Domains can be specified using standard data types, such as integers, floating-point numbers, fixed-length strings, and variant-length strings. The standard data type specifies a broad range of values. Additional constraints can be specified to narrow the ranges. For example, you can specify that the **`numOfCredits`** attribute (in the **`Course`** table) must be greater than 0 and less than 5. If an attribute has different values for each tuple in a relation, you can specify the attribute to be unique. You can also specify whether an attribute can be **`null`**, which is a special value in a database meaning unknown or not applicable. As shown in the **`Student`** table, **`birthDate`** may be **`null`**.

Primary Key Constraints

superkey

A primary key is a set of attributes that uniquely identifyies the tuples in a relations. Why is it called a primary key, rather than simply key? To understand this, it is helpful to know superkeys, keys, and candidate keys. A *superkey* is an attribute or a set of attributes that uniquely identifies the relation. That is, no two tuples have the same values on a superkey. By definition, a relation consists of a set of distinct tuples. The set of all attributes in the relation forms a superkey.

candidate key

primary key

A *key* **`K`** is a minimal superkey, meaning that any proper subset of **`K`** is not a superkey. A relation can have several keys. In this case, each of the keys is called a *candidate key*. The *primary key* is one of the candidate keys designated by the database designer. The primary key is often used to identify tuples in a relation. As shown in Figure 34.6, **`courseId`** is the primary key in the **`Course`** table, and **`ssn`** and **`courseId`** form a primary key in the **`Enrollment`** table.

Foreign Key Constraints

relational database

foreign key constraint

foreign key

In a *relational database*, data are related. Tuples in a relation are related, and tuples in different relations are related through their common attributes. Informally speaking, the common attributes are foreign keys. The *foreign key constraints* define the relationships among relations.

Formally, a set of attributes *FK* is a *foreign key* in a relation *R* that references relation *T* if it satisfies the following two rules:

- The attributes in *FK* have the same domain as the primary key in *T*.
- A nonnull value on *FK* in *R* must match a primary key value in *T*.

As shown in Figure 34.6, `courseId` is the foreign key in `Enrollment` that references the primary key `courseId` in `Course`. Every `courseId` value must match a `courseId` value in `Course`.

Enforcing Integrity Constraints

auto enforcement

The database management system enforces integrity constraints and rejects operations that would violate them. For example, if you attempt to insert the new record (“11115,” “CSCI,” “2490,” “C++ Programming,” “0”) into the `Course` table, it would fail because the credit hours must be greater than `0`; if you attempted to insert a record with the same primary key as an existing record in the table, the DBMS would report an error and reject the operation; if you attempted to delete a record from the `Course` table whose primary key value is referenced by the records in the `Enrollment` table, the DBMS would reject this operation.

Note

All relational database systems support primary key constraints and foreign key constraints, but not all database systems support domain constraints. In the Microsoft Access database, for example, you cannot specify the constraint that `numOfCredits` is greater than `0` and less than `5`.

Check Point

34.2.1 What are superkeys, candidate keys, and primary keys?

34.2.2 What is a foreign key?

34.2.3 Can a relation have more than one primary key or foreign key?

34.2.4 Does a foreign key need to be a primary key in the same relation?

34.2.5 Does a foreign key need to have the same name as its referenced primary key?

34.2.6 Can a foreign key value be null?

34.3 SQL

Key Point

Structured Query Language (SQL) is the language for defining tables and integrity constraints, and for accessing and manipulating data.

SQL

database language

SQL (pronounced “S-Q-L” or “sequel”) is the universal language for accessing relational database systems. Application programs may allow users to access a database without directly using SQL, but these applications themselves must use SQL to access the database. This section introduces some basic SQL commands.

Note

standard SQL

There are many relational database management systems. They share the common SQL language but do not all support every feature of SQL. Some systems have their own extensions to SQL. This section introduces standard SQL supported by all systems.

SQL can be used on MySQL, Oracle, Sybase, IBM DB2, IBM Informix, MS Access, Apache Derby, or any other relational database system. Apache Derby is an open source relational database management system developed using Java. Oracle distributes Apache Derby as Java DB and bundled with Java so you can use it in any Java application without installing a database. Java DB is ideal for supporting a small database in a Java application. This chapter uses MySQL to demonstrate SQL and Java database programming.

The Companion Website contains the following supplements on how to install and use three popular databases: MySQL, Oracle, and Java DB:

MySQL Tutorial

Oracle Tutorial

Java DB Tutorial

- Supplement IV.B: Tutorial for MySQL
- Supplement IV.C: Tutorial for Oracle
- Supplement IV.D: Tutorial for Java DB

34.3.1 Creating a User Account on MySQL

Assume you have installed MySQL 5 with the default configuration. To match all the examples in this book, you should create a user named *scott* with the password *tiger*. You can perform the administrative tasks using the MySQL Workbench or using the command line. MySQL Workbench is a GUI tool for managing MySQL databases. Here are the steps to create a user from the command line:

1. From the DOS command prompt, type

   ```
   mysql –uroot -p
   ```

   You will be prompted to enter the root password, as shown in Figure 34.7.

2. At the mysql prompt, enter

   ```
   use mysql;
   ```

3. To create user **scott** with password **tiger**, enter

   ```
   create user 'scott'@'localhost' identified by 'tiger';
   ```

4. To grant privileges to **scott**, enter

   ```
   grant select, insert, update, delete, create, create view, drop,
      execute, references on *.* to 'scott'@'localhost';
   ```

   - If you want to enable remote access of the account from any IP address, enter

     ```
     grant all privileges on *.* to 'scott'@'%'
       identified by 'tiger';
     ```

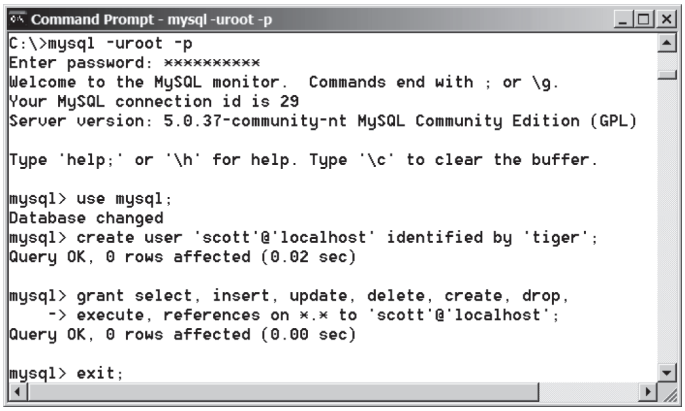
   - If you want to restrict the account's remote access to just one particular IP address, enter

     ```
     grant all privileges on *.* to 'scott'@'ipAddress'
       identified by 'tiger';
     ```

5. Enter

   ```
   exit;
   ```

   to exit the MySQL console.



```
C:\>mysql -uroot -p
Enter password: **********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 29
Server version: 5.0.37-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use mysql;
Database changed
mysql> create user 'scott'@'localhost' identified by 'tiger';
Query OK, 0 rows affected (0.02 sec)

mysql> grant select, insert, update, delete, create, drop,
    -> execute, references on *.* to 'scott'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> exit;
```

FIGURE 34.7 You can access a MySQL database server from the command window.

Note

stop mysql
start mysql

On Windows, your MySQL database server starts every time your computer starts. You can stop it by typing the command `net stop mysql` and restart it by typing the command `net start mysql`.

By default, the server contains two databases named **mysql** and **test**. The **mysql** database contains the tables that store information about the server and its users. This database is intended for the server administrator to use. For example, the administrator can use it to create users and grant or revoke user privileges. Since you are the owner of the server installed on your system, you have full access to the **mysql** database. However, you should not create user tables in the mysql database. You can use the **test** database to store data or create new databases. You can also create a new database using the command `create database` ***`databasename`*** or delete an existing database using the command `drop database` ***`databasename`***.

34.3.2 Creating a Database

To match the examples in this book, you should create a database named **javabook**. Here are the steps to create it:

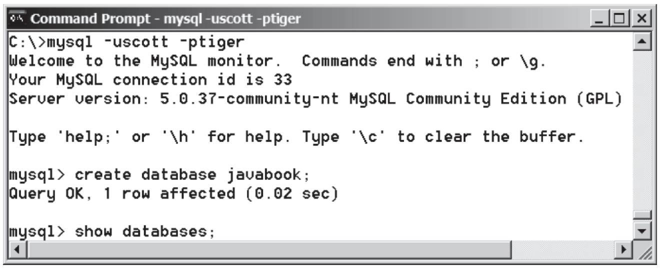
1. From the DOS command prompt, type

   ```
   mysql –uscott -ptiger
   ```

   to login to mysql, as shown in Figure 34.8.

2. At the mysql prompt, enter

   ```
   create database javabook;
   ```



```
C:\>mysql -uscott -ptiger
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 33
Server version: 5.0.37-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database javabook;
Query OK, 1 row affected (0.02 sec)

mysql> show databases;
```

FIGURE 34.8 You can create databases in MySQL.

For your convenience, the SQL statements for creating and initializing tables used in this book are provided in Supplement IV.A. You can download the script for MySQL and save it to **script.sql**. To execute the script, first switch to the **javabook** database using the following command:

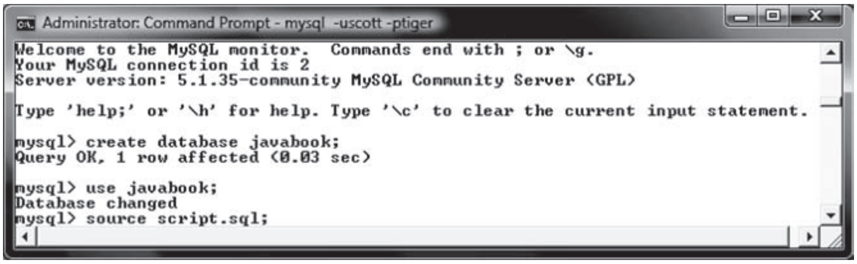
```
use javabook;
```

then type

run script file

```
source script.sql;
```

as shown in Figure 34.9.



```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.1.35-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create database javabook;
Query OK, 1 row affected (0.03 sec)

mysql> use javabook;
Database changed
mysql> source script.sql;
```

FIGURE 34.9 You can run SQL commands in a script file.

Note

You can populate the **javabook** database using the script from Supplement IV.A.

populating database

34.3.3 Creating and Dropping Tables

create table

Tables are the essential objects in a database. To create a table, use the `create table` statement to specify a table name, attributes, and types, as in the following example:

```
create table Course (
  courseId char(5),
  subjectId char(4) not null,
  courseNumber integer,
  title varchar(50) not null,
  numOfCredits integer,
  primary key (courseId)
);
```

This statement creates the `Course` table with attributes `courseId`, `subjectId`, `courseNumber`, `title`, and `numOfCredits`. Each attribute has a data type that specifies the type of data stored in the attribute. `char(5)` specifies that `courseId` consists of five characters. `varchar(50)` specifies that `title` is a variant-length string with a maximum of 50 characters. `integer` specifies that `courseNumber` is an integer. The primary key is `courseId`.

The tables `Student` and `Enrollment` can be created as follows:

```
create table Student (
  ssn char(9),
  firstName varchar(25),
  mi char(1),
  lastName varchar(25),
  birthDate date,
  street varchar(25),
  phone char(11),
  zipCode char(5),
  deptId char(4),
  primary key (ssn)
);
```

```
create table Enrollment (
  ssn char(9),
  courseId char(5),
  dateRegistered date,
  grade char(1),
  primary key (ssn, courseId),
  foreign key (ssn) references
    Student(ssn),
  foreign key (courseId) references
    Course(courseId)
);
```

Note

SQL keywords are not case sensitive. This book adopts the following naming conventions: tables are named in the same way as Java classes, and attributes are named in the same way as Java variables. SQL keywords are named in the same way as Java keywords.

naming convention

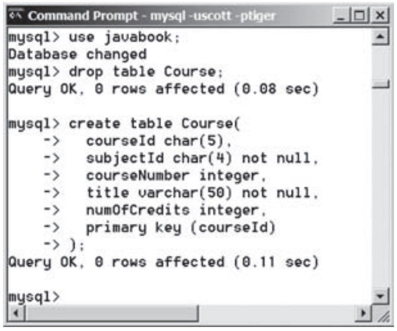
drop table

If a table is no longer needed, it can be dropped permanently using the `drop table` command. For example, the following statement drops the `Course` table:

```
drop table Course;
```

If a table to be dropped is referenced by other tables, you have to drop the other tables first. For example, if you have created the tables `Course`, `Student`, and `Enrollment` and want to drop `Course`, you have to first drop `Enrollment`, because `Course` is referenced by `Enrollment`.

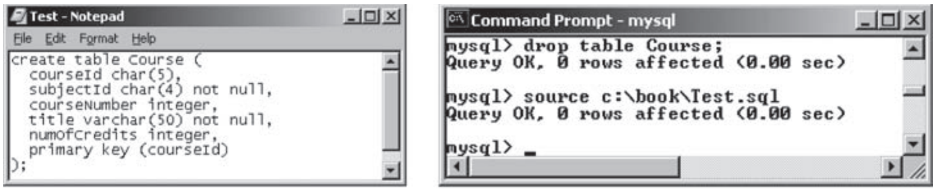
Figure 34.10 shows how to enter the `create table` statement from the MySQL console.



```
mysql> use javabook;
Database changed
mysql> drop table Course;
Query OK, 0 rows affected (0.08 sec)

mysql> create table Course(
    ->   courseId char(5),
    ->   subjectId char(4) not null,
    ->   courseNumber integer,
    ->   title varchar(50) not null,
    ->   numOfCredits integer,
    ->   primary key (courseId)
    -> );
Query OK, 0 rows affected (0.11 sec)

mysql>
```

FIGURE 34.10 A table is created using the `create table` statement.

If you make typing errors, you have to retype the whole command. To avoid retyping, you can save the command in a file, then run the command from the file. To do so, create a text file to contain commands, named, for example, **test.sql**. You can create the text file using any text editor, such as Notepad, as shown in Figure 34.11a. To comment a line, precede it with two dashes. You can now run the script file by typing `source test.sql` from the SQL command prompt, as shown in Figure 34.11b.



```
create table Course (
  courseId char(5),
  subjectId char(4) not null,
  courseNumber integer,
  title varchar(50) not null,
  numOfCredits integer,
  primary key (courseId)
);
```

```
mysql> drop table Course;
Query OK, 0 rows affected (0.00 sec)

mysql> source c:\book\Test.sql
Query OK, 0 rows affected (0.00 sec)

mysql> _
```

(a) (b)

FIGURE 34.11 (a) You can use Notepad to create a text file for SQL commands. (b) You can run the SQL commands in a script file from MySQL.

34.3.4 Simple Insert, Update, and Delete

Once a table is created, you can insert data into it. You can also update and delete records. This section introduces simple insert, update, and delete statements.

The syntax to insert a record into a table is:

```
insert into tableName [(column1, column2, ..., column)]
values (value1, value2, ..., valuen);
```

For example, the following statement inserts a record into the `Course` table. The new record has the `courseId` '11113', `subjectId` 'CSCI', `courseNumber` '3720', `title` 'Database Systems', and `creditHours` 3.

```
insert into Course (courseId, subjectId, courseNumber, title, numOfCredits)
values ('11113', 'CSCI', '3720', 'Database Systems', 3);
```

The column names are optional. If they are omitted, all the column values for the record must be entered, even though the columns have default values. String values are case sensitive and enclosed inside single quotation marks in SQL.

The syntax to update a table is:

```
update tableName
set column1 = newValue1 [, column2 = newValue2, ...]
[where condition];
```

For example, the following statement changes the `numOfCredits` for the course whose `title` is Database Systems to 4.

```
update Course
set numOfCredits = 4
where title = 'Database Systems';
```

The syntax to delete records from a table is:

```
delete from tableName
[where condition];
```

For example, the following statement deletes the Database Systems course from the `Course` table:

```
delete from Course
where title = 'Database Systems';
```

The following statement deletes all the records from the `Course` table:

```
delete from Course;
```

34.3.5 Simple Queries

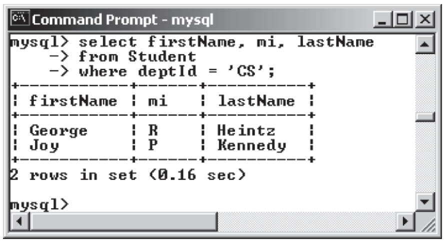
To retrieve information from tables, use a `select` statement with the following syntax:

```
select column-list
from table-list
[where condition];
```

The `select` clause lists the columns to be selected. The `from` clause refers to the tables involved in the query. The optional `where` clause specifies the conditions for the selected rows.

Query 1: Select all the students in the CS department, as shown in Figure 34.12.

```
select firstName, mi, lastName
from Student
where deptId = 'CS';
```



```
mysql> select firstName, mi, lastName
    -> from Student
    -> where deptId = 'CS';
+-----------+------+----------+
| firstName | mi   | lastName |
+-----------+------+----------+
| George    | R    | Heintz   |
| Joy       | P    | Kennedy  |
+-----------+------+----------+
2 rows in set (0.16 sec)

mysql>
```

FIGURE 34.12 The result of the `select` statement is displayed in the MySQL console.

34.3.6 Comparison and Boolean Operators

SQL has six comparison operators, as shown in Table 34.1, and three Boolean operators, as shown in Table 34.2.

TABLE 34.1 Comparison Operators

Operator	*Description*
`=`	Equal to
`<>` or `!=`	Not equal to
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to

TABLE 34.2 Boolean Operators

Operator	*Description*
`not`	Logical negation
`and`	Logical conjunction
`or`	Logical disjunction

Note

The comparison and Boolean operators in SQL have the same meanings as in Java. In SQL the `equal to` operator is `=`, but in Java it is `==`. In SQL the `not equal to` operator is `<>` or `!=`, but in Java it is `!=`. The `not`, `and`, and `or` operators are `!`, `&&` (`&`), and `||` (`|`) in Java.

Query 2: Get the names of the students who are in the CS dept and live in the ZIP code 31411.

```
select firstName, mi, lastName
from Student
where deptId = 'CS' and zipCode = '31411';
```

Note

To select all the attributes from a table, you don't have to list all the attribute names in the select clause. Instead, you can just use an *asterisk* (*), which stands for all the attributes. For example, the following query displays all the attributes of the students who are in the CS dept and live in ZIP code 31411.

```
select *
from Student
where deptId = 'CS' and zipCode = '31411';
```

34.3.7 The `like`, `between-and`, and `is null` Operators

SQL has a `like` operator that can be used for pattern matching. The syntax to check whether a string `s` has a pattern `p` is

`s like p` or `s not like p`

You can use the wildcard characters `%` (percent symbol) and `_` (underline symbol) in the pattern `p`. `%` matches zero or more characters, and `_` matches any single character in `s`. For example, `lastName like '_mi%'` matches any string whose second and third letters are `m` and `i`. `lastName not like '_mi%'` excludes any string whose second and third letters are `m` and `i`.

Note

In earlier versions of MS Access, the wildcard character is `*`, and the character `?` matches any single character.

The `between-and` operator checks whether a value `v` is between two other values, `v1` and `v2`, using the following syntax:

`v between v1 and v2` or `v not between v1 and v2`

`v between v1 and v2` is equivalent to `v >= v1 and v <= v2`, and `v not between v1 and v2` is equivalent to `v < v1 or v > v2`.

The `is null` operator checks whether a value `v` is `null` using the following syntax:

`v is null` or `v is not null`

Query 3: Get the Social Security numbers of the students whose grades are between ‘C’ and ‘A’.

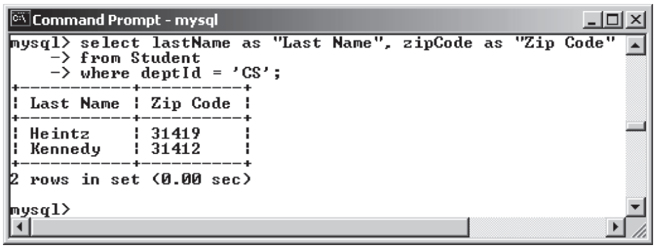
```
select ssn
from Enrollment
where grade between 'C' and 'A';
```

34.3.8 Column Alias

When a query result is displayed, SQL uses the column names as column headings. Usually the user gives abbreviated names for the columns, and the columns cannot have spaces when the table is created. Sometimes it is desirable to give more descriptive names in the result heading. You can use the column aliases with the following syntax:

```
select columnName [as] alias
```

Query 4: Get the last name and ZIP code of the students in the CS department. Display the column headings as “Last Name” for lastName and “Zip Code” for zipCode. The query result is shown in Figure 34.13.



```
mysql> select lastName as "Last Name", zipCode as "Zip Code"
    -> from Student
    -> where deptId = 'CS';
+-----------+----------+
| Last Name | Zip Code |
+-----------+----------+
| Heintz    | 31419    |
| Kennedy   | 31412    |
+-----------+----------+
2 rows in set (0.00 sec)

mysql>
```

FIGURE 34.13 You can use a column alias in the display.

```
select lastName as "Last Name", zipCode as "Zip Code"
from Student
where deptId = 'CS';
```

Note

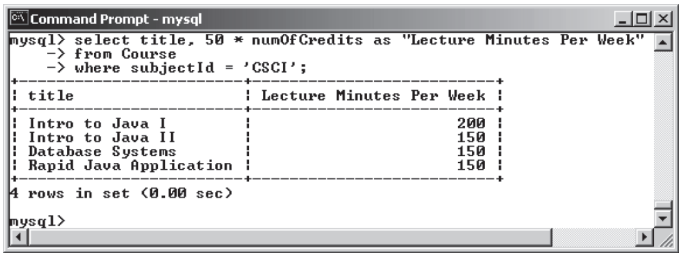
The `as` keyword is optional in MySQL and Oracle, but it is required in MS Access.

34.3.9 The Arithmetic Operators

You can use the arithmetic operators `*` (multiplication), `/` (division), `+` (addition), and `-` (subtraction) in SQL.

Query 5: Assume a credit hour is 50 minutes of lectures and get the total minutes for each course with the subject CSCI. The query result is shown in Figure 34.14.

```
select title, 50 * numOfCredits as "Lecture Minutes Per Week"
from Course
where subjectId = 'CSCI';
```



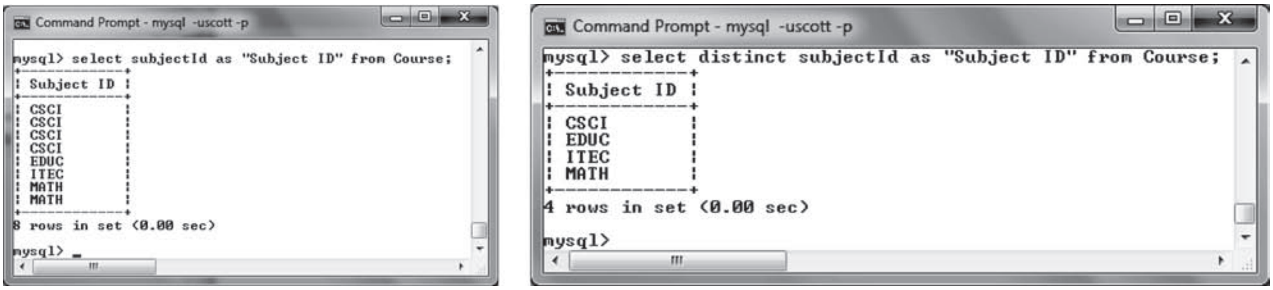
```
mysql> select title, 50 * numOfCredits as "Lecture Minutes Per Week"
    -> from Course
    -> where subjectId = 'CSCI';
+-------------------------+--------------------------+
| title                   | Lecture Minutes Per Week |
+-------------------------+--------------------------+
| Intro to Java I         |                      200 |
| Intro to Java II        |                      150 |
| Database Systems        |                      150 |
| Rapid Java Application  |                      150 |
+-------------------------+--------------------------+
4 rows in set (0.00 sec)

mysql>
```

FIGURE 34.14 You can use arithmetic operators in SQL.

34.3.10 Displaying Distinct Tuples

SQL provides the `distinct` keyword, which can be used to eliminate duplicate tuples in the result. Figure 34.15a displays all the subject IDs used by the courses, and Figure 34.15b displays all the distinct subject IDs used by the courses using the following statement:

```
select distinct subjectId as "Subject ID"
from Course;
```


Command Prompt - mysql -uscott -p
Command Prompt - mysql -uscott -p

```
mysql> select subjectId as "Subject ID" from Course;
+------------+
| Subject ID |
+------------+
| CSCI       |
| CSCI       |
| CSCI       |
| CSCI       |
| EDUC       |
| ITEC       |
| MATH       |
| MATH       |
+------------+
8 rows in set (0.00 sec)

mysql> _
```

(a)

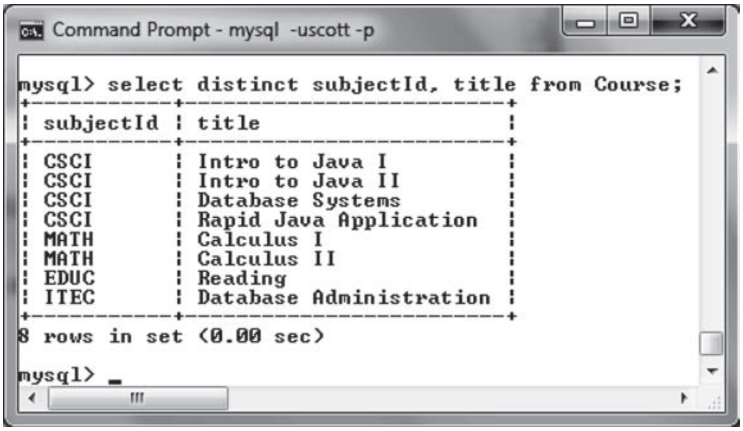
```
mysql> select distinct subjectId as "Subject ID" from Course;
+------------+
| Subject ID |
+------------+
| CSCI       |
| EDUC       |
| ITEC       |
| MATH       |
+------------+
4 rows in set (0.00 sec)

mysql>
```

(b)

FIGURE 34.15 (a) The duplicate tuples are displayed. (b) The distinct tuples are displayed.

When there is more than one column in the `select` clause, the `distinct` keyword applies to the whole tuple in the result. For example, the following statement displays all tuples with distinct `subjectId` and `title`, as shown in Figure 34.16. Note some tuples may have the same `subjectId` but different `title`. These tuples are distinct.

```
select distinct subjectId, title
from Course;
```



```
mysql> select distinct subjectId, title from Course;
+-----------+----------------------------+
| subjectId | title                      |
+-----------+----------------------------+
| CSCI      | Intro to Java I            |
| CSCI      | Intro to Java II           |
| CSCI      | Database Systems           |
| CSCI      | Rapid Java Application     |
| MATH      | Calculus I                 |
| MATH      | Calculus II                |
| EDUC      | Reading                    |
| ITEC      | Database Administration    |
+-----------+----------------------------+
8 rows in set (0.00 sec)

mysql> _
```

FIGURE 34.16 The keyword `distinct` applies to the entire tuple.

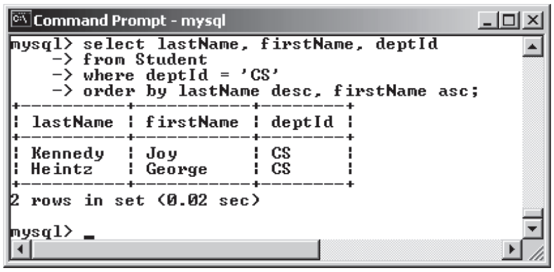
34.3.11 Displaying Sorted Tuples

SQL provides the `order by` clause to sort the output using the following syntax:

```
select column-list
from table-list
[where condition]
[order by columns-to-be-sorted];
```

In the syntax, `columns-to-be-sorted` specifies a column or a list of columns to be sorted. By default, the order is ascending. To sort in a descending order, append the `desc` keyword. You could also append the `asc` keyword after `columns-to-be-sorted`, but it is not necessary. When multiple columns are specified, the rows are sorted based on the first column, then the rows with the same values on the first column are sorted based on the second column, and so on.

Query 6: List the full names of the students in the CS department, ordered primarily on their last names in descending order and secondarily on their first names in ascending order. The query result is shown in Figure 34.17.



```
mysql> select lastName, firstName, deptId
    -> from Student
    -> where deptId = 'CS'
    -> order by lastName desc, firstName asc;
+----------+-----------+--------+
| lastName | firstName | deptId |
+----------+-----------+--------+
| Kennedy  | Joy       | CS     |
| Heintz   | George    | CS     |
+----------+-----------+--------+
2 rows in set (0.02 sec)

mysql> _
```

FIGURE 34.17 You can sort results using the `order by` clause.

```
select lastName, firstName, deptId
from Student
where deptId = 'CS'
order by lastName desc, firstName asc;
```

34.3.12 Joining Tables

Often you need to get information from multiple tables, as demonstrated in the next query.

Query 7: List the courses taken by the student Jacob Smith. To solve this query, you need to join tables `Student` and `Enrollment`, as shown in Figure 34.18.

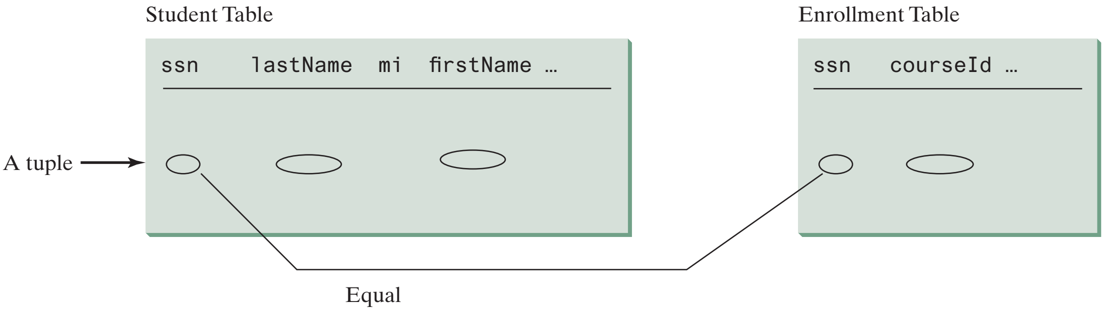
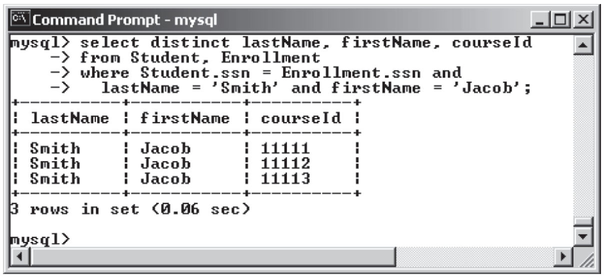


FIGURE 34.18 `Student` and `Enrollment` are joined on `ssn`.

You can write the query in SQL as follows:

```
select distinct lastName, firstName, courseId
from Student, Enrollment
where Student.ssn = Enrollment.ssn and
  lastName = 'Smith' and firstName = 'Jacob';
```

The tables `Student` and `Enrollment` are listed in the `from` clause. The query examines every pair of rows, each made of one item from `Student` and another from `Enrollment` and selects the pairs that satisfy the condition in the `where` clause. The rows in `Student` have the last name, Smith, and the first name, Jacob, and both rows from `Student` and `Enrollment` have the same `ssn` values. For each pair selected, `lastName` and `firstName` from `Student` and `courseId` from `Enrollment` are used to produce the result, as shown in Figure 34.19. `Student` and `Enrollment` have the same attribute `ssn`. To distinguish them in a query, use `Student.ssn` and `Enrollment.ssn`.



```
mysql> select distinct lastName, firstName, courseId
    -> from Student, Enrollment
    -> where Student.ssn = Enrollment.ssn and
    ->   lastName = 'Smith' and firstName = 'Jacob';
+----------+-----------+----------+
| lastName | firstName | courseId |
+----------+-----------+----------+
| Smith    | Jacob     | 11111    |
| Smith    | Jacob     | 11112    |
| Smith    | Jacob     | 11113    |
+----------+-----------+----------+
3 rows in set (0.06 sec)

mysql>
```

FIGURE 34.19 Query 7 demonstrates queries involving multiple tables.

For more features of SQL, see Supplements IV.H and IV.I.

Check Point

34.3.1 Create the tables `Course`, `Student`, and `Enrollment` using the `create table` statements in Section 34.3.3, Creating and Dropping Tables. Insert rows into the `Course`, `Student`, and `Enrollment` tables using the data in Figures 34.3–34.5.

34.3.2 List all CSCI courses with at least four credit hours.

34.3.3 List all students whose last names contain the letter *e* two times.

34.3.4 List all students whose birthdays are null.

34.3.5 List all students who take Math courses.

34.3.6 List the number of courses in each subject.

34.3.7 Assume each credit hour is 50 minutes of lectures. Get the total minutes for the courses that each student takes.

34.4 JDBC

JDBC is the Java API for accessing relational database.

Key Point

The Java API for developing Java database applications is called *JDBC*. JDBC is the trademarked name of a Java API that supports Java programs that access relational databases. JDBC is not an acronym, but it is often thought to stand for Java Database Connectivity.

JDBC provides Java programmers with a uniform interface for accessing and manipulating relational databases. Using the JDBC API, applications written in the Java programming language can execute SQL statements, retrieve results, present data in a user-friendly interface, and propagate changes back to the database. The JDBC API can also be used to interact with multiple data sources in a distributed, heterogeneous environment.

The relationships among Java programs, JDBC API, JDBC drivers, and relational databases are shown in Figure 34.20. The JDBC API is a set of Java interfaces and classes used to write Java programs for accessing and manipulating relational databases. Since a JDBC driver serves as the interface to facilitate communications between JDBC and a proprietary database, JDBC drivers are database specific and are normally provided by the database vendors. You need

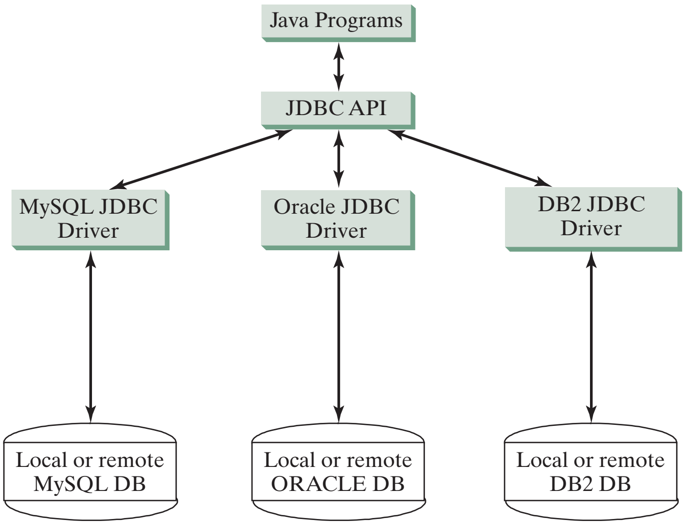


FIGURE 34.20 Java programs access and manipulate databases through JDBC drivers.

MySQL JDBC drivers to access the MySQL database, Oracle JDBC drivers to access the Oracle database, and DB2 JDBC driver to access the DB2 database.

34.4.1 Developing Database Applications Using JDBC

The JDBC API is a Java application program interface to generic SQL databases that enables Java developers to develop DBMS-independent Java applications using a uniform interface.

The JDBC API consists of classes and interfaces for establishing connections with databases, sending SQL statements to databases, processing the results of SQL statements, and obtaining database metadata. Four key interfaces are needed to develop any database application using Java: `Driver`, `Connection`, `Statement`, and `ResultSet`. These interfaces define a framework for generic SQL database access. The JDBC API defines these interfaces, and the JDBC driver vendors provide the implementation for the interfaces. Programmers use these interfaces.

The relationship of these interfaces is shown in Figure 34.21. A JDBC application loads an appropriate driver using the `Driver` interface, connects to the database using the `Connection` interface, creates and executes SQL statements using the `Statement` interface, and processes the result using the `ResultSet` interface if the statements return results. Note some statements, such as SQL data definition statements and SQL data modification statements, do not return results.

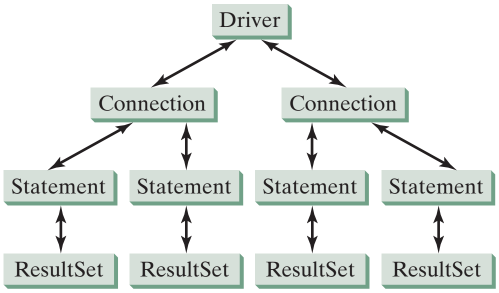


FIGURE 34.21 JDBC classes enable Java programs to connect to the database, send SQL statements, and process results.

The JDBC interfaces and classes are the building blocks in the development of Java database programs. A typical Java program takes the following steps to access a database.

1. Loading drivers.

An appropriate driver must be loaded using the statement shown below before connecting to a database.

```
Class.forName("JDBCDriverClass");
```

A driver is a concrete class that implements the `java.sql.Driver` interface. The drivers for MySQL, Oracle, and Java DB are listed in Table 34.3. If your program accesses several different databases, all their respective drivers must be loaded.

mysql-connector-java-5.1.26.jar
ojdbc6.jar

The most recent platform independent version of MySQL JDBC driver is **mysql-connector-java-5.1.26.jar.** This file is contained in a ZIP file downloadable from dev.mysql.com/downloads/connector/j/. The most recent version of Oracle JDBC driver is **ojdbc6.jar** (downloadable from www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html).

TABLE 34.3 JDBC Drivers

Database	*Driver Class*	*Source*
MySQL	`com.mysql.jdbc.Driver`	**mysql-connector-java-5.1.26.jar**
Oracle	`oracle.jdbc.driver.OracleDriver`	**ojdbc6.jar**
Java DB (embedded)	`org.apache.derby.jdbc.EmbeddedDriver`	**derby.jar**
Java DB (network)	`org.apache.derby.jdbc.ClientDriver`	**derbynet.jar**

Java DB has two versions: embedded and networked. Embedded version is used when you access Java DB locally, while the network version enables you to access Java DB on the network. To use these drivers, you have to add their jar files in the classpath using the following DOS command on Windows:

```
set classpath=%classpath%;c:\book\lib\mysql-connector-java-5.1.26.
jar;c:\book\lib\ojdbc6.jar;c:\program files\jdk1.8.0\db\lib\derby.
jar
```

If you use an IDE such as Eclipse or NetBeans, you need to add these jar files into the library in the IDE.

why load a driver?

Note

`com.mysql.jdbc.Driver` is a class in `mysql-connector-java-5.1.26.jar`, and `oracle.jdbc.driver.OracleDriver` is a class in `ojdbc6.jar`. `mysql-connector-java-5.1.26.jar`, `ojdbc6.jar`, and `derby.jar` contains many classes to support the driver. These classes are used by JDBC but not directly by JDBC programmers. When you use a class explicitly in the program, it is automatically loaded by the JVM. The driver classes, however, are not used explicitly in the program, so you have to write the code to tell the JVM to load them.

automatic driver discovery

Note

Java supports automatic driver discovery, so you don't have to load the driver explicitly. At the time of this writing, however, this feature is not supported for all database drivers. To be safe, load the driver explicitly.

2. Establishing connections.

To connect to a database, use the static method `getConnection(databaseURL)` in the `DriverManager` class, as follows:

```
Connection connection = DriverManager.getConnection(databaseURL);
```

where `databaseURL` is the unique identifier of the database on the Internet. Table 34.4 lists the URL patterns for the MySQL, Oracle, and Java DB.

TABLE 34.4 JDBC URLs

Database	*URL Pattern*
MySQL	`jdbc:mysql://hostname/dbname`
Oracle	`jdbc:oracle:thin:@hostname:port#:oracleDBSID`
Java DB (embedded)	`jdbc:derby:dbname`
Java DB (network)	`jdbc:derby://hostname/dbname`

connect MySQL DB

The `databaseURL` for a MySQL database specifies the host name and database name to locate a database. For example, the following statement creates a `Connection` object for the local MySQL database `javabook` with username *scott* and password *tiger*:

```
Connection connection = DriverManager.getConnection
  ("jdbc:mysql://localhost/javabook", "scott", "tiger");
```

Recall that by default, MySQL contains two databases named *mysql* and *test*. Section 34.3.2, Creating a Database, created a custom database named `javabook`. We will use `javabook` in the examples.

connect Oracle DB

The `databaseURL` for an Oracle database specifies the *hostname*, the *port#* where the database listens for incoming connection requests, and the *oracleDBSID* database name to locate a database. For example, the following statement creates a `Connection` object for the Oracle database on liang.armstrong.edu with the username *scott* and password *tiger*:

```
Connection connection = DriverManager.getConnection
  ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
   "scott", "tiger");
```

3. Creating statements.

If a `Connection` object can be envisioned as a cable linking your program to a database, an object of `Statement` can be viewed as a cart that delivers SQL statements for execution by the database and brings the result back to the program. Once a `Connection` object is created, you can create statements for executing SQL statements as follows:

```
Statement statement = connection.createStatement();
```

4. Executing statements.

SQL data definition language (DDL) and update statements can be executed using `executeUpdate(String sql)`, and an SQL query statement can be executed using `executeQuery(String sql)`. The result of the query is returned in `ResultSet`. For example, the following code executes the SQL statement `create table Temp (col1 char(5), col2 char(5))`:

```
statement.executeUpdate
  ("create table Temp (col1 char(5), col2 char(5))");
```

This next code executes the SQL query `select firstName, mi, lastName from Student where lastName = 'Smith'`:

```
// Select the columns from the Student table
ResultSet resultSet = statement.executeQuery
  ("select firstName, mi, lastName from Student where lastName "
    + " = 'Smith'");
```

5. Processing `ResultSet`.

The `ResultSet` maintains a table whose current row can be retrieved. The initial row position is `null`. You can use the `next` method to move to the next row and the various getter methods to retrieve values from a current row. For example, the following code displays all the results from the preceding SQL query:

```
// Iterate through the result and print the student names
while (resultSet.next())
  System.out.println(resultSet.getString(1) + " " +
    resultSet.getString(2) + " " + resultSet.getString(3));
```

The `getString(1)`, `getString(2)`, and `getString(3)` methods retrieve the column values for `firstName`, `mi`, and `lastName`, respectively. Alternatively, you can use `getString("firstName")`, `getString("mi")`, and `getString("lastName")` to retrieve the same three column values. The first execution of the `next()` method sets the current row to the first row in the result set, and subsequent invocations of the `next()` method set the current row to the second row, third row, and so on, to the last row.

Listing 34.1 is a complete example that demonstrates connecting to a database, executing a simple query, and processing the query result with JDBC. The program connects to a local MySQL database and displays the students whose last name is `Smith`.

LISTING 34.1 SimpleJdbc.java

```
 1  import java.sql.*;
 2
 3  public class SimpleJdbc {
 4   public static void main(String[] args)
 5     throws SQLException, ClassNotFoundException {
 6    // Load the JDBC driver
 7    Class.forName("com.mysql.jdbc.Driver");
 8    System.out.println("Driver loaded");
 9
10    // Connect to a database
11    Connection connection = DriverManager.getConnection
12      ("jdbc:mysql://localhost/javabook", "scott", "tiger");
13    System.out.println("Database connected");
14
15    // Create a statement
16    Statement statement = connection.createStatement();
17
18    // Execute a statement
19    ResultSet resultSet = statement.executeQuery
20      ("select firstName, mi, lastName from Student where lastName "
21      + " = 'Smith'");
22
23    // Iterate through the result and print the student names
24    while (resultSet.next())
25      System.out.println(resultSet.getString(1) + "\t" +
26        resultSet.getString(2) + "\t" + resultSet.getString(3));
27
28    // Close the connection
29    connection.close();
30   }
31  }
```

load driver

connect database

create statement

execute statement

get result

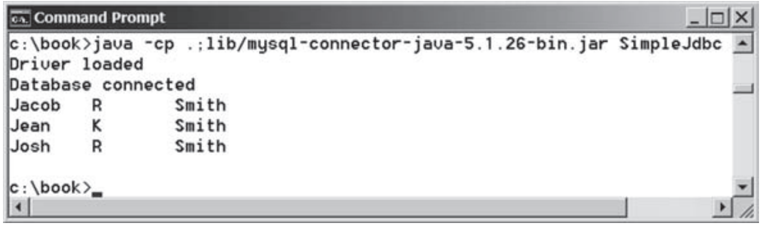
close connection

The statement in line 7 loads a JDBC driver for MySQL, and the statement in lines 11–13 connects to a local MySQL database. You can change them to connect to an Oracle or other databases. The program creates a `Statement` object (line 16), executes an SQL statement and returns a `ResultSet` object (lines 19–21), and retrieves the query result from the `ResultSet` object (lines 24–26). The last statement (line 29) closes the connection and releases resources related to the connection. You can rewrite this program using the try-with-resources syntax. See www.cs.armstrong.edu/liang/intro11e/html/SimpleJdbcWithAutoClose.html.

Note

If you run this program from the DOS prompt, specify the appropriate driver in the classpath, as shown in Figure 34.22.

run from DOS prompt



```
c:\book>java -cp .;lib/mysql-connector-java-5.1.26-bin.jar SimpleJdbc
Driver loaded
Database connected
Jacob    R        Smith
Jean     K        Smith
Josh     R        Smith

c:\book>
```

FIGURE 34.22 You must include the driver file to run Java database programs.

The classpath directory and jar files are separated by commas. The period (`.`) represents the current directory. For convenience, the driver files are placed under the **lib** directory.

the semicolon issue

Caution

Do not use a semicolon (`;`) to end the Oracle SQL command in a Java program. The semicolon may not work with the Oracle JDBC drivers. It does work, however, with the other drivers used in this book.

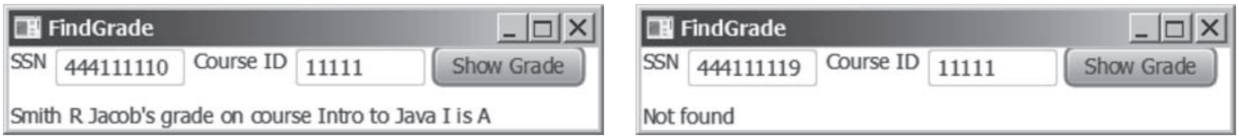
autocommit

Note

The `Connection` interface handles transactions and specifies how they are processed. By default, a new connection is in autocommit mode, and all its SQL statements are executed and committed as individual transactions. The commit occurs when the statement completes or the next execute occurs, whichever comes first. In the case of statements returning a result set, the statement completes when the last row of the result set has been retrieved or the result set has been closed. If a single statement returns multiple results, the commit occurs when all the results have been retrieved. You can use the `setAutoCommit(false)` method to disable autocommit, so all SQL statements are grouped into one transaction that is terminated by a call to either the `commit()` or the `rollback()` method. The `rollback()` method undoes all the changes made by the transaction.

34.4.2 Accessing a Database from JavaFX

This section gives an example that demonstrates connecting to a database from a JavaFX program. The program lets the user enter the SSN and the course ID to find a student's grade, as shown in Figure 34.23. The code in Listing 34.2 uses the MySQL database on the localhost.



SSN 444111110 Course ID 11111

Smith R Jacob's grade on course Intro to Java I is A

SSN 444111119 Course ID 11111

Not found

FIGURE 34.23 A JavaFX client can access the database on the server.

LISTING 34.2 FindGrade.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import java.sql.*;
```

```java
public class FindGrade extends Application {
  // Statement for executing queries
  private Statement stmt;
  private TextField tfSSN = new TextField();
  private TextField tfCourseId = new TextField();
  private Label lblStatus = new Label();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Initialize database connection and create a Statement object
    initializeDB();

    Button btShowGrade = new Button("Show Grade");
    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(new Label("SSN"), tfSSN,
      new Label("Course ID"), tfCourseId, (btShowGrade));

    VBox vBox = new VBox(10);
    vBox.getChildren().addAll(hBox, lblStatus);

    tfSSN.setPrefColumnCount(6);
    tfCourseId.setPrefColumnCount(6);
    btShowGrade.setOnAction(e -> showGrade());

    // Create a scene and place it in the stage
    Scene scene = new Scene(vBox, 420, 80);
    primaryStage.setTitle("FindGrade"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  private void initializeDB() {
    try {
      // Load the JDBC driver
      Class.forName("com.mysql.jdbc.Driver");
//      Class.forName("oracle.jdbc.driver.OracleDriver");
      System.out.println("Driver loaded");

      // Establish a connection
      Connection connection = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook", "scott", "tiger");
//    ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
//     "scott", "tiger");
      System.out.println("Database connected");

      // Create a statement
      stmt = connection.createStatement();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  private void showGrade() {
    String ssn = tfSSN.getText();
    String courseId = tfCourseId.getText();
    try {
      String queryString = "select firstName, mi, " +
        "lastName, title, grade from Student, Enrollment, Course " +
```

button listener

load driver

Oracle driver commented

connect to MySQL database

connect to Oracle commented

execute statement

show result

create statement

```
70         "where Student.ssn = '" + ssn + "' and Enrollment.courseId "
71         + "= '" + courseId +
72         "' and Enrollment.courseId = Course.courseId " +
73         " and Enrollment.ssn = Student.ssn";
74
75       ResultSet rset = stmt.executeQuery(queryString);
76
77       if (rset.next()) {
78         String lastName = rset.getString(1);
79         String mi = rset.getString(2);
80         String firstName = rset.getString(3);
81         String title = rset.getString(4);
82         String grade = rset.getString(5);
83
84         // Display result in a label
85         lblStatus.setText(firstName + " " + mi +
86           " " + lastName + "'s grade on course " + title + " is " +
87           grade);
88       } else {
89         lblStatus.setText("Not found");
90       }
91     }
92     catch (SQLException ex) {
93       ex.printStackTrace();
94     }
95   }
96 }
```

The `initializeDB()` method (lines 42–62) loads the MySQL driver (line 45), connects to the MySQL database on host `liang.armstrong.edu` (lines 50–55), and creates a statement (line 57).

security hole

Note

There is a *security hole* in this program. If you enter `1' or true or '1` in the `SSN` field, you will get the first student's score, because the query string now becomes

```
select firstName, mi, lastName, title, grade
from Student, Enrollment, Course
where Student.ssn = '1' or true or '1' and
      Enrollment.courseId = ' ' and
      Enrollment.courseId = Course.courseId and
      Enrollment.ssn = Student.ssn;
```

You can avoid this problem by using the `PreparedStatement` interface, which will be discussed in the next section.

Check Point

34.4.1 What are the advantages of developing database applications using Java?

34.4.2 Describe the following JDBC interfaces: `Driver`, `Connection`, `Statement`, and `ResultSet`.

34.4.3 How do you load a JDBC driver? What are the driver classes for MySQL, Oracle, and Java DB?

34.4.4 How do you create a database connection? What are the URLs for MySQL, Oracle, and Java DB?

34.4.5 How do you create a `Statement` and execute an SQL statement?

34.4.6 How do you retrieve values in a `ResultSet`?

34.4.7 Does JDBC automatically commit a transaction? How do you set autocommit to false?

34.5 PreparedStatement

Key Point

`PreparedStatement` enables you to create parameterized SQL statements.

Once a connection to a particular database is established, it can be used to send SQL statements from your program to the database. The `Statement` interface is used to execute static SQL statements that don't contain any parameters. The `PreparedStatement` interface, extending `Statement`, is used to execute a precompiled SQL statement with or without parameters. Since the SQL statements are precompiled, they are efficient for repeated executions.

A `PreparedStatement` object is created using the `prepareStatement` method in the `Connection` interface. For example, the following code creates a `PreparedStatement` for an SQL `insert` statement:

```
PreparedStatement preparedStatement = connection.prepareStatement
  ("insert into Student (firstName, mi, lastName) " +
   "values (?, ?, ?)");
```

This `insert` statement has three question marks as placeholders for parameters representing values for `firstName`, `mi`, and `lastName` in a record of the `Student` table.

As a subinterface of `Statement`, the `PreparedStatement` interface inherits all the methods defined in `Statement`. It also provides the methods for setting parameters in the object of `PreparedStatement`. These methods are used to set the values for the parameters before executing statements or procedures. In general, the setter methods have the following name and signature:

```
setX(int parameterIndex, X value);
```

where *X* is the type of the parameter, and `parameterIndex` is the index of the parameter in the statement. The index starts from `1`. For example, the method `setString(int parameterIndex, String value)` sets a `String` value to the specified parameter.

The following statements pass the parameters `"Jack"`, `"A"`, and `"Ryan"` to the placeholders for `firstName`, `mi`, and `lastName` in `preparedStatement`:

```
preparedStatement.setString(1, "Jack");
preparedStatement.setString(2, "A");
preparedStatement.setString(3, "Ryan");
```

After setting the parameters, you can execute the prepared statement by invoking `executeQuery()` for a SELECT statement and `executeUpdate()` for a DDL or update statement.

The `executeQuery()` and `executeUpdate()` methods are similar to the ones defined in the `Statement` interface except that they don't have any parameters, because the SQL statements are already specified in the `prepareStatement` method when the object of `PreparedStatement` is created.

Using a prepared SQL statement, Listing 34.2 can be improved as in Listing 34.3.

LISTING 34.3 FindGradeUsingPreparedStatement.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import java.sql.*;

```

```
public class FindGradeUsingPreparedStatement extends Application {
  // PreparedStatement for executing queries
  private PreparedStatement preparedStatement;
  private TextField tfSSN = new TextField();
  private TextField tfCourseId = new TextField();
  private Label lblStatus = new Label();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Initialize database connection and create a Statement object
    initializeDB();

    Button btShowGrade = new Button("Show Grade");
    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(new Label("SSN"), tfSSN,
      new Label("Course ID"), tfCourseId, (btShowGrade));

    VBox vBox = new VBox(10);
    vBox.getChildren().addAll(hBox, lblStatus);

    tfSSN.setPrefColumnCount(6);
    tfCourseId.setPrefColumnCount(6);
    btShowGrade.setOnAction(e -> showGrade());

```

prepare statement

```
    // Create a scene and place it in the stage
    Scene scene = new Scene(vBox, 420, 80);
    primaryStage.setTitle("FindGrade"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  private void initializeDB() {
    try {
      // Load the JDBC driver
```

load driver

```
      Class.forName("com.mysql.jdbc.Driver");
//      Class.forName("oracle.jdbc.driver.OracleDriver");
      System.out.println("Driver loaded");

      // Establish a connection
```

connect database

```
      Connection connection = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook", "scott", "tiger");
//    ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
//     "scott", "tiger");
      System.out.println("Database connected");

      String queryString = "select firstName, mi, " +
        "lastName, title, grade from Student, Enrollment, Course " +
```

placeholder

```
        "where Student.ssn = ? and Enrollment.courseId = ? " +
        "and Enrollment.courseId = Course.courseId";

      // Create a statement
      preparedStatement = connection.prepareStatement(queryString);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  private void showGrade() {
    String ssn = tfSSN.getText();
```

```
71        String courseId = tfCourseId.getText();
72        try {
73          preparedStatement.setString(1, ssn);
74          preparedStatement.setString(2, courseId);
75          ResultSet rset = preparedStatement.executeQuery();
76
77          if (rset.next()) {
78            String lastName = rset.getString(1);
79            String mi = rset.getString(2);
80            String firstName = rset.getString(3);
81            String title = rset.getString(4);
82            String grade = rset.getString(5);
83
84            // Display result in a label
85            lblStatus.setText(firstName + " " + mi +
86              " " + lastName + "'s grade on course " + title + " is " +
87              grade);
88          } else {
89            lblStatus.setText("Not found");
90          }
91        }
92        catch (SQLException ex) {
93          ex.printStackTrace();
94        }
95      }
96    }
```

execute statement

show result

This example does exactly the same thing as Listing 34.2 except that it uses the prepared statement to dynamically set the parameters. The code in this example is almost the same as in the preceding example. The new code is highlighted.

A prepared query string is defined in lines 56–59 with `ssn` and `courseId` as parameters. An SQL prepared statement is obtained in line 62. Before executing the query, the actual values of `ssn` and `courseId` are set to the parameters in lines 73–74. Line 75 executes the prepared statement.

Check Point

34.5.1 Describe prepared statements. How do you create instances of `PreparedStatement`? How do you execute a `PreparedStatement`? How do you set parameter values in a `PreparedStatement`?

34.5.2 What are the benefits of using prepared statements?

34.6 CallableStatement

Key Point

`CallableStatement` enables you to execute SQL stored procedures.

The `CallableStatement` interface is designed to execute SQL-stored procedures. The procedures may have `IN`, `OUT`, or `IN OUT` parameters. An `IN` parameter receives a value passed to the procedure when it is called. An `OUT` parameter returns a value after the procedure is completed, but it doesn't contain any value when the procedure is called. An `IN OUT` parameter contains a value passed to the procedure when it is called and returns a value after it is completed. For example, the following procedure in Oracle PL/SQL has `IN` parameter `p1`, `OUT` parameter `p2`, and `IN OUT` parameter `p3`:

IN parameter
OUT parameter
IN OUT parameter

```
create or replace procedure sampleProcedure
  (p1 in varchar, p2 out number, p3 in out integer) is
begin
  /* do something */
end sampleProcedure;
/
```

Note

The syntax of stored procedures is vendor specific. We use both Oracle and MySQL for demonstrations of stored procedures in this book.

A `CallableStatement` object can be created using the `prepareCall(String call)` method in the `Connection` interface. For example, the following code creates a `CallableStatement cstmt` on `Connection connection` for the procedure `sampleProcedure`:

```
CallableStatement callableStatement = connection.prepareCall(
  "{call sampleProcedure(?, ?, ?)}");
```

`{call sampleProcedure(?, ?, ...)}` is referred to as the *SQL escape syntax*, which signals the driver that the code within it should be handled differently. The driver parses the escape syntax and translates it into code that the database understands. In this example, `sampleProcedure` is an Oracle procedure. The call is translated to the string `begin sampleProcedure(?, ?, ?); end` and passed to an Oracle database for execution.

You can call procedures as well as functions. The syntax to create an SQL callable statement for a function is:

```
{? = call functionName(?, ?, ...)}
```

`CallableStatement` inherits `PreparedStatement`. Additionally, the `CallableStatement` interface provides methods for registering the `OUT` parameters and for getting values from the `OUT` parameters.

Before calling an SQL procedure, you need to use appropriate setter methods to pass values to `IN` and `IN OUT` parameters, and use `registerOutParameter` to register `OUT` and `IN OUT` parameters. For example, before calling procedure `sampleProcedure`, the following statements pass values to parameters `p1` (`IN`) and `p3` (`IN OUT`) and register parameters `p2` (`OUT`) and `p3` (`IN OUT`):

```
callableStatement.setString(1, "Dallas"); // Set Dallas to p1
callableStatement.setLong(3, 1); // Set 1 to p3
// Register OUT parameters
callableStatement.registerOutParameter(2, java.sql.Types.DOUBLE);
callableStatement.registerOutParameter(3, java.sql.Types.INTEGER);
```

You can use `execute()` or `executeUpdate()` to execute the procedure depending on the type of SQL statement, then use getter methods to retrieve values from the `OUT` parameters. For example, the next statements retrieve the values from parameters `p2` and `p3`:

```
double d = callableStatement.getDouble(2);
int i = callableStatement.getInt(3);
```

Let us define a MySQL function that returns the number of the records in the table that match the specified `firstName` and `lastName` in the `Student` table.

```
/* For the callable statement example. Use MySQL version 5 */
drop function if exists studentFound;

delimiter //

create function studentFound(first varchar(20), last varchar(20))
  returns int
begin
  declare result int;

  select count(*) into result
  from Student
  where Student.firstName = first and
    Student.lastName = last;

  return result;
end;
//

delimiter ;
/* Please note that there is a space between delimiter and ; */
```

If you use an Oracle database, the function can be defined as follows:

```
create or replace function studentFound
  (first varchar2, last varchar2)
  /* Do not name firstName and lastName. */
  return number is numberOfSelectedRows number := 0;
begin
  select count(*) into numberOfSelectedRows
  from Student
  where Student.firstName = first and
    Student.lastName = last;

  return numberOfSelectedRows;
end studentFound;
/
```

Suppose the function `studentFound` is already created in the database. Listing 34.4 gives an example that tests this function using callable statements.

Listing 34.4 TestCallableStatement.java

```
import java.sql.*;

public class TestCallableStatement {
  /** Creates new form TestTableEditor */
  public static void main(String[] args) throws Exception {
    Class.forName("com.mysql.jdbc.Driver");
    Connection connection = DriverManager.getConnection(
      "jdbc:mysql://localhost/javabook",
      "scott", "tiger");
//    Connection connection = DriverManager.getConnection(
//      ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
//      "scott", "tiger");

    // Create a callable statement
    CallableStatement callableStatement = connection.prepareCall(
      "{? = call studentFound(?, ?)}");

    java.util.Scanner input = new java.util.Scanner(System.in);
    System.out.print("Enter student's first name: ");
    String firstName = input.nextLine();
    System.out.print("Enter student's last name: ");
    String lastName = input.nextLine();

    callableStatement.setString(2, firstName);
    callableStatement.setString(3, lastName);
    callableStatement.registerOutParameter(1, Types.INTEGER);
    callableStatement.execute();
```

load driver
connect database
create callable statement
enter `firstName`
enter `lastName`
set IN parameter
set IN parameter
register OUT parameter
execute statement

get OUT parameter

```
28
29      if (callableStatement.getInt(1) >= 1)
30        System.out.println(firstName + " " + lastName +
31          " is in the database");
32      else
33        System.out.println(firstName + " " + lastName +
34          " is not in the database");
35    }
36  }
```

```
Enter student's first name: Jacob ↵Enter
Enter student's last name: Smith ↵Enter
Jacob Smith is in the database
```

```
Enter student's first name: John ↵Enter
Enter student's last name: Smith ↵Enter
John Smith is not in the database
```

The program loads a MySQL driver (line 6), connects to a MySQL database (lines 7–9), and creates a callable statement for executing the function `studentFound` (lines 15–16).

The function's first parameter is the return value; its second and third parameters correspond to the first and last names. Before executing the callable statement, the program sets the first name and last name (lines 24–25) and registers the `OUT` parameter (line 26). The statement is executed in line 27.

The function's return value is obtained in line 29. If the value is greater than or equal to `1`, the student with the specified first and last name is found in the table.

Check Point

34.6.1 Describe callable statements. How do you create instances of `CallableStatement`? How do you execute a `CallableStatement`? How do you register `OUT` parameters in a `CallableStatement`?

34.7 Retrieving Metadata

Key Point

The database metadata such as database URL, username, and JDBC driver name can be obtained using the `DatabaseMetaData` *interface and result set metadata such as table column count and column names can be obtained using the* `ResultSetMetaData` *interface.*

database metadata

JDBC provides the `DatabaseMetaData` interface for obtaining database-wide information, and the `ResultSetMetaData` interface for obtaining information on a specific `ResultSet`.

34.7.1 Database Metadata

The `Connection` interface establishes a connection to a database. It is within the context of a connection that SQL statements are executed and results are returned. A connection also provides access to database metadata information that describes the capabilities of the database, supported SQL grammar, stored procedures, and so on. To obtain an instance of `DatabaseMetaData` for a database, use the `getMetaData` method on a `Connection` object like this:

```
DatabaseMetaData dbMetaData = connection.getMetaData();
```

If your program connects to a local MySQL database, the program in Listing 34.5 displays the database information statements shown in Figure 34.24.

LISTING 34.5 `TestDatabaseMetaData.java`

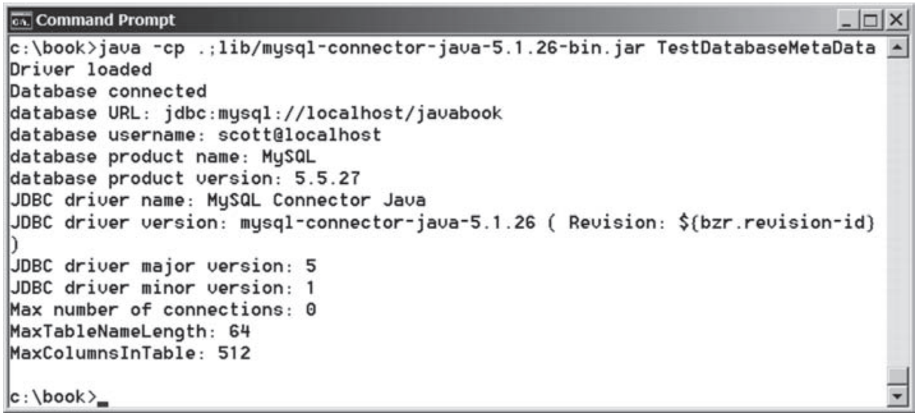
```java
import java.sql.*;

public class TestDatabaseMetaData {
  public static void main(String[] args)
      throws SQLException, ClassNotFoundException {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

    // Connect to a database
    Connection connection = DriverManager.getConnection
      ("jdbc:mysql://localhost/javabook", "scott", "tiger");
    System.out.println("Database connected");

    DatabaseMetaData dbMetaData = connection.getMetaData();
    System.out.println("database URL: " + dbMetaData.getURL());
    System.out.println("database username: " +
      dbMetaData.getUserName());
    System.out.println("database product name: " +
      dbMetaData.getDatabaseProductName());
    System.out.println("database product version: " +
      dbMetaData.getDatabaseProductVersion());
    System.out.println("JDBC driver name: " +
      dbMetaData.getDriverName());
    System.out.println("JDBC driver version: " +
      dbMetaData.getDriverVersion());
    System.out.println("JDBC driver major version: " +
      dbMetaData.getDriverMajorVersion());
    System.out.println("JDBC driver minor version: " +
      dbMetaData.getDriverMinorVersion());
    System.out.println("Max number of connections: " +
      dbMetaData.getMaxConnections());
    System.out.println("MaxTableNameLength: " +
      dbMetaData.getMaxTableNameLength());
    System.out.println("MaxColumnsInTable: " +
      dbMetaData.getMaxColumnsInTable());

    // Close the connection
    connection.close();
  }
}
```

load driver

connect database

database metadata

get metadata



```
c:\book>java -cp .;lib/mysql-connector-java-5.1.26-bin.jar TestDatabaseMetaData
Driver loaded
Database connected
database URL: jdbc:mysql://localhost/javabook
database username: scott@localhost
database product name: MySQL
database product version: 5.5.27
JDBC driver name: MySQL Connector Java
JDBC driver version: mysql-connector-java-5.1.26 ( Revision: ${bzr.revision-id}
)
JDBC driver major version: 5
JDBC driver minor version: 1
Max number of connections: 0
MaxTableNameLength: 64
MaxColumnsInTable: 512

c:\book>_
```

FIGURE 34.24 The `DatabaseMetaData` interface enables you to obtain database information.

34.7.2 Obtaining Database Tables

You can identify the tables in the database through database metadata using the `getTables` method. Listing 34.6 displays all the user tables in the javabook database on a local MySQL database. Figure 34.25 shows a sample output of the program.

LISTING 34.6 `FindUserTables.java`

load driver

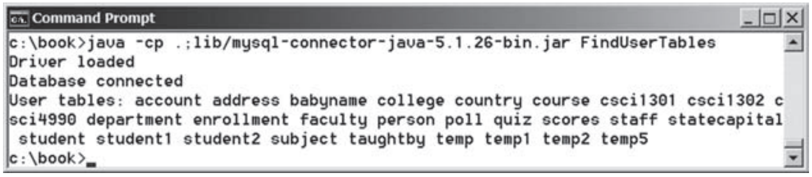
connect database

database metadata

obtain tables

get table names

```
 1  import java.sql.*;
 2
 3  public class FindUserTables {
 4    public static void main(String[] args)
 5        throws SQLException, ClassNotFoundException {
 6      // Load the JDBC driver
 7      Class.forName("com.mysql.jdbc.Driver");
 8      System.out.println("Driver loaded");
 9
10      // Connect to a database
11      Connection connection = DriverManager.getConnection
12        ("jdbc:mysql://localhost/javabook", "scott", "tiger");
13      System.out.println("Database connected");
14
15      DatabaseMetaData dbMetaData = connection.getMetaData();
16
17      ResultSet rsTables = dbMetaData.getTables(null, null, null,
18        new String[] {"TABLE"});
19      System.out.print("User tables: ");
20      while (rsTables.next())
21        System.out.print(rsTables.getString("TABLE_NAME") + " ");
22
23      // Close the connection
24      connection.close();
25    }
26  }
```



```
c:\book>java -cp .;lib/mysql-connector-java-5.1.26-bin.jar FindUserTables
Driver loaded
Database connected
User tables: account address babyname college country course csci1301 csci1302 c
sci4990 department enrollment faculty person poll quiz scores staff statecapital
 student student1 student2 subject taughtby temp temp1 temp2 temp5
c:\book>_
```

FIGURE 34.25 You can find all the tables in the database.

Line 17 obtains table information in a result set using the `getTables` method. One of the columns in the result set is TABLE_NAME. Line 21 retrieves the table name from this result set column.

34.7.3 Result Set Metadata

The `ResultSetMetaData` interface describes information pertaining to the result set. A `ResultSetMetaData` object can be used to find the types and properties of the columns in a `ResultSet`. To obtain an instance of `ResultSetMetaData`, use the `getMetaData` method on a result set like this:

```
ResultSetMetaData rsMetaData = resultSet.getMetaData();
```

You can use the `getColumnCount()` method to find the number of columns in the result and the `getColumnName(int)` method to get the column names. For example, Listing 34.7 displays all the column names and contents resulting from the SQL SELECT statement *`select * from Enrollment`*. The output is shown in Figure 34.26.

Listing 34.7 TestResultSetMetaData.java

```java
import java.sql.*;

public class TestResultSetMetaData {
  public static void main(String[] args)
      throws SQLException, ClassNotFoundException {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

    // Connect to a database
    Connection connection = DriverManager.getConnection
      ("jdbc:mysql://localhost/javabook", "scott", "tiger");
    System.out.println("Database connected");

    // Create a statement
    Statement statement = connection.createStatement();

    // Execute a statement
    ResultSet resultSet = statement.executeQuery
      ("select * from Enrollment");

    ResultSetMetaData rsMetaData = resultSet.getMetaData();
    for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
      System.out.printf("%-12s\t", rsMetaData.getColumnName(i));
    System.out.println();

    // Iterate through the result and print the students' names
    while (resultSet.next()) {
      for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
        System.out.printf("%-12s\t", resultSet.getObject(i));
      System.out.println();
    }

    // Close the connection
    connection.close();
  }
}
```

load driver

connect database

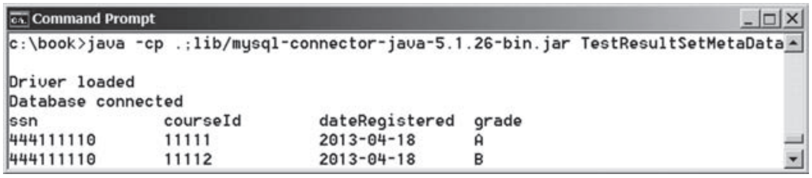
create statement

create result set

result set metadata

column count

column name



```
c:\book>java -cp .;lib/mysql-connector-java-5.1.26-bin.jar TestResultSetMetaData

Driver loaded
Database connected
ssn             courseId        dateRegistered  grade
444111110       11111           2013-04-18      A
444111110       11112           2013-04-18      B
```

Figure 34.26 The `ResultSetMetaData` interface enables you to obtain result set information.

Check Point

34.7.1 What is `DatabaseMetaData` for? Describe the methods in `DatabaseMetaData`. How do you get an instance of `DatabaseMetaData`?

34.7.2 What is `ResultSetMetaData` for? Describe the methods in `ResultSetMetaData`. How do you get an instance of `ResultSetMetaData`?

34.7.3 How do you find the number of columns in a result set? How do you find the column names in a result set?

KEY TERMS

candidate key	34-5
database system	34-2
domain constraint	34-5
foreign key	34-5
foreign key constraint	34-5
integrity constraint	34-4
primary key	34-5
relational database	34-5
Structured Query Language (SQL)	34-6
superkey	34-5

CHAPTER SUMMARY

1. This chapter introduced the concepts of *database systems*, *relational databases*, *relational data models*, *data integrity*, and *SQL*. You learned how to develop database applications using Java.

2. The Java API for developing Java database applications is called *JDBC*. JDBC provides Java programmers with a uniform interface for accessing and manipulating relational databases.

3. The JDBC API consists of classes and interfaces for establishing connections with databases, sending SQL statements to databases, processing the results of SQL statements, and obtaining database metadata.

4. Since a JDBC driver serves as the interface to facilitate communications between JDBC and a proprietary database, JDBC drivers are database specific. If you use a driver, make sure it is in the classpath before running the program.

5. Four key interfaces are needed to develop any database application using Java: `Driver`, `Connection`, `Statement`, and `ResultSet`. These interfaces define a framework for generic SQL database access. The JDBC driver vendors provide implementation for them.

6. A JDBC application loads an appropriate driver using the `Driver` interface, connects to the database using the `Connection` interface, creates and executes SQL statements using the `Statement` interface, and processes the result using the `ResultSet` interface if the statements return results.

7. The `PreparedStatement` interface is designed to execute dynamic SQL statements with parameters. These SQL statements are precompiled for efficient use when repeatedly executed.

8. Database *metadata* is information that describes the database itself. JDBC provides the `DatabaseMetaData` interface for obtaining database-wide information and the `ResultSetMetaData` interface for obtaining information on the specific `ResultSet`.

Quiz

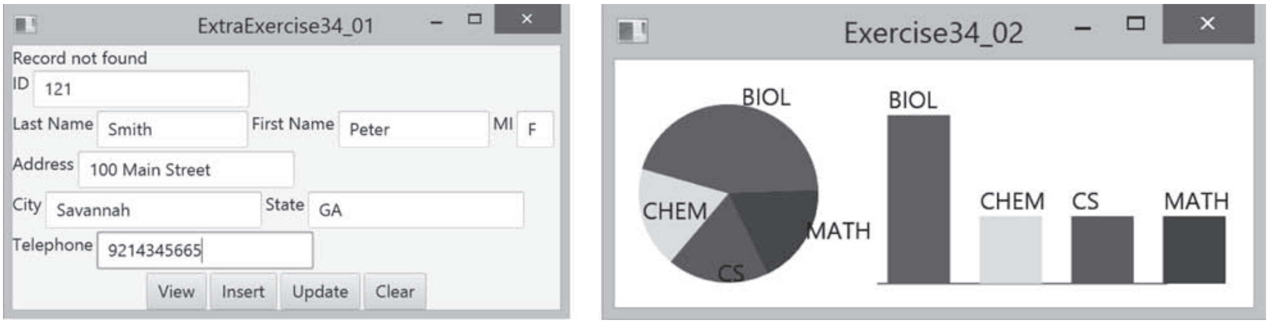
Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

***34.1** (*Access and update a* `Staff` *table*) Write a program that views, inserts, and updates staff information stored in a database, as shown in Figure 34.27a. The *View* button displays a record with a specified ID. The *Insert* button inserts a new record. The *Update* button updates the record for the specified ID. The `Staff` table is created as follows:

```
create table Staff (
  id char(9) not null,
  lastName varchar(15),
  firstName varchar(15),
  mi char(1),
  address varchar(20),
  city varchar(20),
  state char(2),
  telephone char(10),
  email varchar(40),
  primary key (id)
);
```



ExtraExercise34_01

Record not found

ID: 121

Last Name: Smith

First Name: Peter

MI: F

Address: 100 Main Street

City: Savannah

State: GA

Telephone: 9214345665

(a)

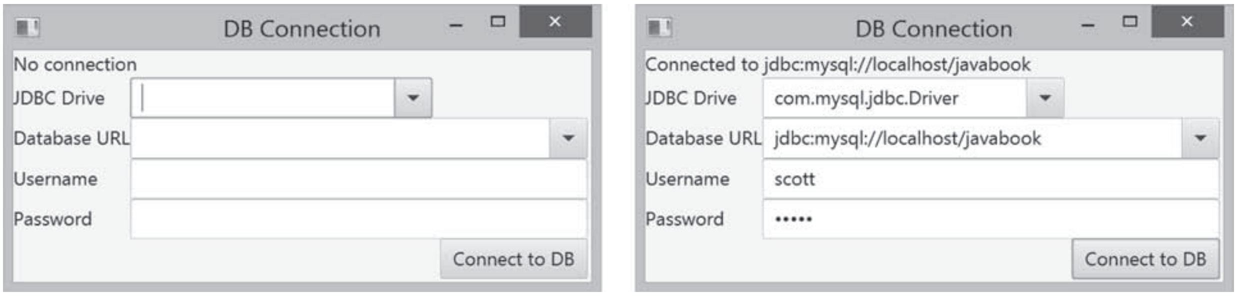
(b)

Figure 34.27 (a) The program lets you view, insert, and update staff information. (b) The `PieChart` and `BarChart` components display the query data obtained from the data module.

****34.2** (*Visualize data*) Write a program that displays the number of students in each department in a pie chart and a bar chart, as shown in Figure 34.27b. The `PieChart` and `BarChart` classes are created in Programming Exercises 14.12 and 14.13. The number of students for each department can be obtained from the `Student` table (see Figure 34.4) using the following SQL statement:

```
select deptId, count(*)
from Student
where deptId is not null
group by deptId;
```

***34.3** (*Connection dialog*) Develop a subclass of `BorderPane` named `DBConnectionPane` that enables the user to select or enter a JDBC driver and a URL and to enter a username and password, as shown in Figure 34.28. When the *Connect to DB* button is clicked, a `Connection` object for the database is stored in the `connection` property. You can then use the `getConnection()` method to return the connection.

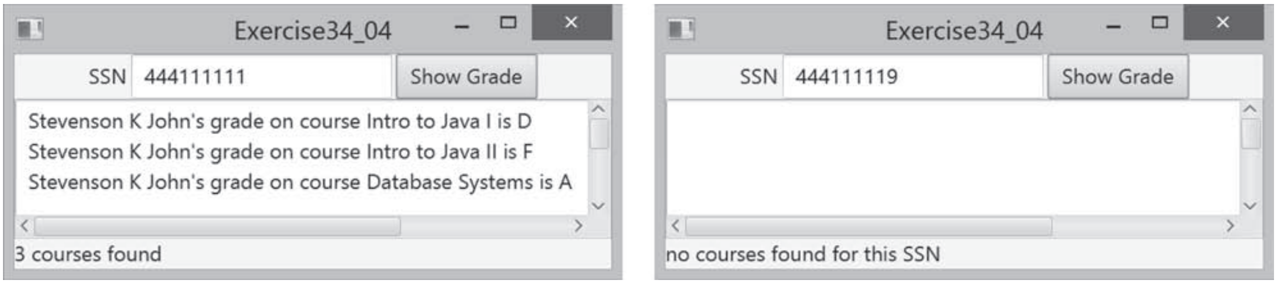


No connection
JDBC Drive
Database URL
Username
Password

Connected to jdbc:mysql://localhost/javabook
JDBC Drive: com.mysql.jdbc.Driver
Database URL: jdbc:mysql://localhost/javabook
Username: scott
Password: •••••

FIGURE 34.28 The `DBConnectionPane` component enables the user to enter database information.

***34.4** (*Find grades*) Listing 34.2, FindGrade.java, presented a program that finds a student's grade for a specified course. Rewrite the program to find all the grades for a specified student, as shown in Figure 34.29.



SSN 444111111

```
Stevenson K John's grade on course Intro to Java I is D
Stevenson K John's grade on course Intro to Java II is F
Stevenson K John's grade on course Database Systems is A
```

3 courses found

SSN 444111119

no courses found for this SSN

FIGURE 34.29 The program displays the grades for the courses for a specified student.

***34.5** (*Display table contents*) Write a program that displays the content for a given table. As shown in Figure 34.30a, you enter a table and click the *Show Contents* button to display the table contents in the text area.

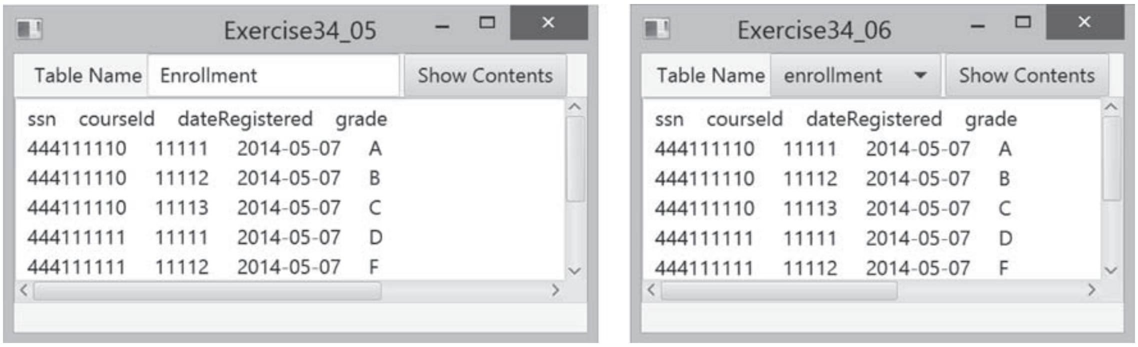


Table Name Enrollment

```
ssn       courseId  dateRegistered  grade
444111110  11111   2014-05-07   A
444111110  11112   2014-05-07   B
444111110  11113   2014-05-07   C
444111111  11111   2014-05-07   D
444111111  11112   2014-05-07   F
```

Table Name enrollment

```
ssn       courseId  dateRegistered  grade
444111110  11111   2014-05-07   A
444111110  11112   2014-05-07   B
444111110  11113   2014-05-07   C
444111111  11111   2014-05-07   D
444111111  11112   2014-05-07   F
```

(a)

(b)

FIGURE 34.30 (a) Enter a table name to display the table contents. (b) Select a table name from the combo box to display its contents.

***34.6** (*Find tables and showing their contents*) Write a program that fills in table names in a combo box, as shown in Figure 34.30b. You can select a table from the combo box to display its contents in the text area.

****34.7** (*Populate Quiz table*) Create a table named `Quiz` as follows:

```
create table Quiz(
  questionId int,
  question varchar(4000),
  choicea varchar(1000),
  choiceb varchar(1000),
  choicec varchar(1000),
  choiced varchar(1000),
  answer varchar(5));
```

The **`Quiz`** table stores multiple-choice questions. Suppose the multiple-choice questions are stored in a text file accessible from http://www.cs.armstrong.edu/liang/data/Quiz.txt in the following format:

```
1. question1
a. choice a
b. choice b
c. choice c
d. choice d
Answer:cd

2. question2
a. choice a
b. choice b
c. choice c
d. choice d
Answer:a

...
```

Write a program that reads the data from the file and populate it into the **`Quiz`** table.

***34.8** (*Populate Salary table*) Create a table named **`Salary`** as follows:

```
create table Salary(
  firstName varchar(100),
  lastName varchar(100),
  rank varchar(15),
  salary float);
```

Obtain the data for salary from http://cs.armstrong.edu/liang/data/Salary.txt and populate it into the **`Salary`** table in the database.

***34.9** (*Copy table*) Suppose the database contains a student table defined as follows:

```
create table Student1 (
  username varchar(50) not null,
  password varchar(50) not null,
  fullname varchar(200) not null,
  constraint pkStudent primary key (username)
);
```

Create a new table named **`Student2`** as follows:

```
create table Student2 (
  username varchar(50) not null,
  password varchar(50) not null,
  firstname varchar(100),
  lastname varchar(100),
  constraint pkStudent primary key (username)
);
```

A full name is in the form of **`firstname mi lastname`** or **`firstname lastname`**. For example, **`John K Smith`** is a full name. Write a program that copies table `Student1` into `Student2`. Your task is to split a full name into `firstname`, `mi`, and `lastname` for each record in `Student1` and store a new record into `Student2`.

***34.10** (*Record unsubmitted exercises*) The following three tables store information on students, assigned exercises, and exercise submission in LiveLab. LiveLab is an automatic grading system for grading programming exercises.

```
create table AGSStudent (
  username varchar(50) not null,
  password varchar(50) not null,
  fullname varchar(200) not null,
  instructorEmail varchar(100) not null,
  constraint pkAGSStudent primary key (username)
);

 create table ExerciseAssigned (
  instructorEmail varchar(100),
  exerciseName varchar(100),
  maxscore double default 10,
  constraint pkCustomExercise primary key
    (instructorEmail, exerciseName)
);

create table AGSLog (
  username varchar(50), /* This is the student's user name */
  exerciseName varchar(100), /* This is the exercise */
  score double default null,
  submitted bit default 0,
  constraint pkLog primary key (username, exerciseName)
);
```

The `AGSStudent` table stores the student information. The `ExerciseAssigned` table assigns the exercises by an instructor. The `AGSLog` table stores the grading results. When a student submits an exercise, a record is stored in the `AGSLog` table. However, there is no record in `AGSLog` if a student did not submit the exercise.

Write a program that adds a new record for each student and an assigned exercise to the student in the `AGSLog` table if a student has not submitted the exercise. The record should have `0` on `score` and `submitted`. For example, if the tables contain the following data in `AGSLog` before you run this program, the `AGSLog` table now contains the new records after the program runs.

AGSStudent

username	password	fullname	instructorEmail
abc	p1	John Roo	t@gmail.com
cde	p2	Yao Mi	c@gmail.com
wbc	p3	F3	t@gmail.com

ExerciseAssigned

instructorEmail	exerciseName	maxScore
t@gmail.com	e1	10
t@gmail.com	e2	10
c@gmail.com	e1	4
c@gmail.com	e4	20

AGSLog

username	exerciseName	score	submitted
abc	e1	9	1
wbc	e2	7	1

AGSLog after the program runs

username	exerciseName	score	submitted
abc	e1	9	1
wbc	e2	7	1
abc	e2		0
wbc	e1		0
cde	e1		0
cde	e4		0

***34.11** (*Baby names*) Create the following table:

```
create table Babyname (
  year integer,
  name varchar(50),
  gender char(1),
  count integer,
  constraint pkBabyname primary key (year, name, gender)
);
```

The baby name ranking data was described in Programming Exercise 12.31. Write a program to read data from the following URL and store into the `Babyname` table.
`https://liveexample.pearsoncmg.com/data/babynamesranking2001.txt`,

. . .

`https://liveexample.pearsoncmg.com/data/babynamesranking2010.txt`.

CHAPTER

35

ADVANCED JAVA DATABASE PROGRAMMING

Objectives

- To create a universal SQL client for accessing local or remote database (§35.2).
- To execute SQL statements in a batch mode (§35.3).
- To process updatable and scrollable result sets (§35.4).
- To simplify Java database programming using `RowSet` (§35.5).
- To store and retrieve images in JDBC (§35.6).

35.1 Introduction

Key Point

This chapter introduces advanced features for Java database programming.

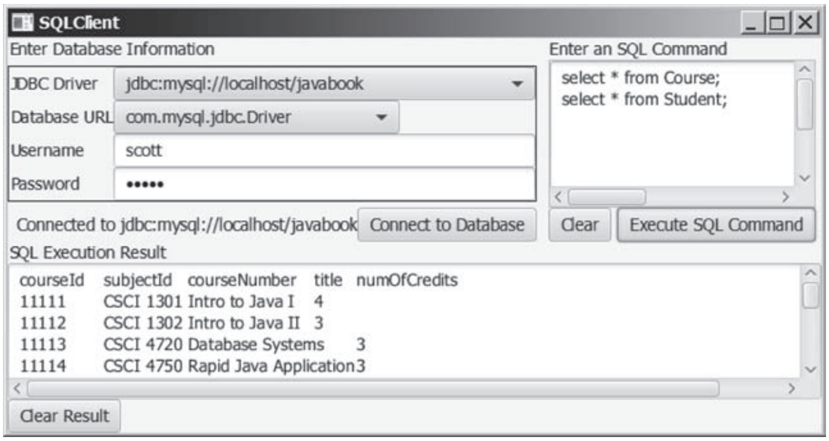
Chapter 34 introduced JDBC's basic features. This chapter covers its advanced features. You will learn how to develop a universal SQL client for accessing any local or remote relational database, learn how to execute statements in a batch mode to improve performance, learn scrollable result sets and how to update a database through result sets, learn how to use `RowSet` to simplify database access, and learn how to store and retrieve images.

35.2 A Universal SQL Client

Key Point

This section develops a universal SQL client for connecting and accessing any SQL database.

In Chapter 34, you used various drivers to connect to the database, created statements for executing SQL statements, and processed the results from SQL queries. This section presents a universal SQL client that enables you to connect to any relational database and execute SQL commands interactively, as shown in Figure 35.1. The client can connect to any JDBC data source and can submit SQL SELECT commands and non-SELECT commands for execution. The execution result is displayed for the SELECT queries, and the execution status is displayed for the non-SELECT commands. Listing 35.1 gives the program.



JDBC Driver: jdbc:mysql://localhost/javabook

Database URL: com.mysql.jdbc.Driver

Username: scott

Password: •••••

```
select * from Course;
select * from Student;
```

Connected to jdbc:mysql://localhost/javabook

```
courseId   subjectId   courseNumber   title   numOfCredits
11111      CSCI 1301 Intro to Java I    4
11112      CSCI 1302 Intro to Java II   3
11113      CSCI 4720 Database Systems       3
11114      CSCI 4750 Rapid Java Application3
```

FIGURE 35.1 You can connect to any JDBC data source and execute SQL commands interactively.

LISTING 35.1 `SQLClient.java`

```java
import java.sql.*;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class SQLClient extends Application {
  // Connection to the database
  private Connection connection;

  // Statement to execute SQL commands
  private Statement statement;

  // Text area to enter SQL commands
  private TextArea tasqlCommand = new TextArea();

  // Text area to display results from SQL commands
  private TextArea taSQLResult = new TextArea();

  // DBC info for a database connection
  private TextField tfUsername = new TextField();
  private PasswordField pfPassword = new PasswordField();
  private ComboBox<String> cboURL = new ComboBox<>();
  private ComboBox<String> cboDriver = new ComboBox<>();

  private Button btExecuteSQL = new Button("Execute SQL Command");
  private Button btClearSQLCommand = new Button("Clear");
  private Button btConnectDB = new Button("Connect to Database");
  private Button btClearSQLResult = new Button("Clear Result");
  private Label lblConnectionStatus
    = new Label("No connection now");

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    cboURL.getItems().addAll(FXCollections.observableArrayList(
      "jdbc:mysql://localhost/javabook",
      "jdbc:mysql://liang.armstrong.edu/javabook",
      "jdbc:odbc:exampleMDBDataSource",
      "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"));
    cboURL.getSelectionModel().selectFirst();

    cboDriver.getItems().addAll(FXCollections.observableArrayList(
     "com.mysql.jdbc.Driver", "sun.jdbc.odbc.dbcOdbcDriver",
     "oracle.jdbc.driver.OracleDriver"));
    cboDriver.getSelectionModel().selectFirst();

    // Create UI for connecting to the database
    GridPane gridPane = new GridPane();
    gridPane.add(cboURL, 1, 0);
    gridPane.add(cboDriver, 1, 1);
    gridPane.add(tfUsername, 1, 2);
    gridPane.add(pfPassword, 1, 3);
    gridPane.add(new Label("JDBC Driver"), 0, 0);
    gridPane.add(new Label("Database URL"), 0, 1);
    gridPane.add(new Label("Username"), 0, 2);
    gridPane.add(new Label("Password"), 0, 3);

    HBox hBoxConnection = new HBox();
    hBoxConnection.getChildren().addAll(
      lblConnectionStatus, btConnectDB);
    hBoxConnection.setAlignment(Pos.CENTER_RIGHT);

    VBox vBoxConnection = new VBox(5);
    vBoxConnection.getChildren().addAll(
      new Label("Enter Database Information"),
      gridPane, hBoxConnection);

    gridPane.setStyle("-fx-border-color: black;");

    HBox hBoxSQLCommand = new HBox(5);
    hBoxSQLCommand.getChildren().addAll(
      btClearSQLCommand, btExecuteSQL);
    hBoxSQLCommand.setAlignment(Pos.CENTER_RIGHT);

    BorderPane borderPaneSqlCommand = new BorderPane();
    borderPaneSqlCommand.setTop(
      new Label("Enter an SQL Command"));
    borderPaneSqlCommand.setCenter(
      new ScrollPane(tasqlCommand));
    borderPaneSqlCommand.setBottom(
      hBoxSQLCommand);

    HBox hBoxConnectionCommand = new HBox(10);
    hBoxConnectionCommand.getChildren().addAll(
      vBoxConnection, borderPaneSqlCommand);

    BorderPane borderPaneExecutionResult = new BorderPane();
    borderPaneExecutionResult.setTop(
      new Label("SQL Execution Result"));
    borderPaneExecutionResult.setCenter(taSQLResult);
    borderPaneExecutionResult.setBottom(btClearSQLResult);

    BorderPane borderPane = new BorderPane();
    borderPane.setTop(hBoxConnectionCommand);
    borderPane.setCenter(borderPaneExecutionResult);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 670, 400);
    primaryStage.setTitle("SQLClient"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    btConnectDB.setOnAction(e -> connectToDB());
    btExecuteSQL.setOnAction(e -> executeSQL());
    btClearSQLCommand.setOnAction(e -> tasqlCommand.setText(null));
    btClearSQLResult.setOnAction(e -> taSQLResult.setText(null));
  }

  /** Connect to DB */
  private void connectToDB() {
    // Get database information from the user input
    String driver = cboDriver
      .getSelectionModel().getSelectedItem();
    String url = cboURL.getSelectionModel().getSelectedItem();
    String username = tfUsername.getText().trim();
    String password = pfPassword.getText().trim();

    // Connection to the database
    try {
      Class.forName(driver);
      connection = DriverManager.getConnection(
        url, username, password);
      lblConnectionStatus.setText("Connected to " + url);
    }
    catch (java.lang.Exception ex) {
      ex.printStackTrace();
    }
  }

  /** Execute SQL commands */
  private void executeSQL() {
    if (connection == null) {
      taSQLResult.setText("Please connect to a database first");
      return;
    }
    else {
      String sqlCommands = tasqlCommand.getText().trim();
      String[] commands = sqlCommands.replace('\n', ' ').split(";");

      for (String aCommand: commands) {
        if (aCommand.trim().toUpperCase().startsWith("SELECT")) {
          processSQLSelect(aCommand);
        }
        else {
          processSQLNonSelect(aCommand);
        }
      }
    }
  }

  /** Execute SQL SELECT commands */
  private void processSQLSelect(String sqlCommand) {
    try {
      // Get a new statement for the current connection
      statement = connection.createStatement();

      // Execute a SELECT SQL command
      ResultSet resultSet = statement.executeQuery(sqlCommand);

      // Find the number of columns in the result set
      int columnCount = resultSet.getMetaData().getColumnCount();
      String row = "";

      // Display column names
      for (int i = 1; i <= columnCount; i++) {
        row += resultSet.getMetaData().getColumnName(i) + "\t";
      }

      taSQLResult.appendText(row + '\n');

      while (resultSet.next()) {
        // Reset row to empty
        row = "";

        for (int i = 1; i <= columnCount; i++) {
          // A non-String column is converted to a string
          row += resultSet.getString(i) + "\t";
        }

        taSQLResult.appendText(row + '\n');
      }
    }
    catch (SQLException ex) {
      taSQLResult.setText(ex.toString());
    }
  }

```

```
200    /** Execute SQL DDL, and modification commands */
201    private void processSQLNonSelect(String sqlCommand) {
202      try {
203        // Get a new statement for the current connection
204        statement = connection.createStatement();
205
206        // Execute a non-SELECT SQL command
207        statement.executeUpdate(sqlCommand);
208
209        taSQLResult.setText("SQL command executed");
210      }
211      catch (SQLException ex) {
212        taSQLResult.setText(ex.toString());
213      }
214    }
215  }
```

The user selects or enters the JDBC driver, database URL, username, and password, and clicks the *Connect to Database* button to connect to the specified database using the `connectToDB()` method (lines 122–140).

When the user clicks the *Execute SQL Command* button, the `executeSQL()` method is invoked (lines 143–161) to get the SQL commands from the text area (`jtaSQLCommand`) and extract each command separated by a semicolon (`;`). It then determines whether the command is a SELECT query or a DDL or data modification statement (lines 153–158). If the command is a SELECT query, the `processSQLSelect` method is invoked (lines 164–198). This method uses the `executeQuery` method (line 170) to obtain the query result. The result is displayed in the text area `jtaSQLResult` (line 181). If the command is a non-SELECT query, the `processSQLNonSelect()` method is invoked (lines 201–214). This method uses the `executeUpdate` method (line 207) to execute the SQL command.

The `getMetaData` method (lines 173, 178) in the `ResultSet` interface is used to obtain an instance of `ResultSetMetaData`. The `getColumnCount` method (line 173) returns the number of columns in the result set, and the `getColumnName(i)` method (line 178) returns the column name for the *i*th column.

35.3 Batch Processing

Key Point

You can send a batch of SQL statements to the database for execution at once to improve efficiency.

In all the preceding examples, SQL commands are submitted to the database for execution one at a time. This is inefficient for processing a large number of updates. For example, suppose you wanted to insert a thousand rows into a table. Submitting one INSERT command at a time would take nearly a thousand times longer than submitting all the INSERT commands in a batch at once. To improve performance, JDBC introduced the batch update for processing nonselect SQL commands. A batch update consists of a sequence of nonselect SQL commands. These commands are collected in a batch and submitted to the database all together.

To use the batch update, you add nonselect commands to a batch using the `addBatch` method in the `Statement` interface. After all the SQL commands are added to the batch, use the `executeBatch` method to submit the batch to the database for execution.

For example, the following code adds a create table command, adds two insert statements in a batch, and executes the batch:

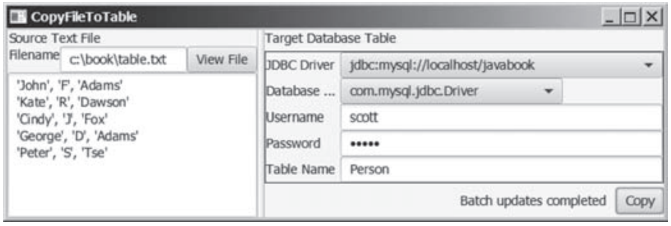
```
Statement statement = connection.createStatement();

// Add SQL commands to the batch
statement.addBatch("create table T (C1 integer, C2 varchar(15))");
statement.addBatch("insert into T values (100, 'Smith')");
statement.addBatch("insert into T values (200, 'Jones')");

// Execute the batch
int count[] = statement.executeBatch();
```

The `executeBatch()` method returns an array of counts, each of which counts the number of rows affected by the SQL command. The first count returns `0` because it is a DDL command. The other counts return `1` because only one row is affected.

> **Note**
>
> To find out whether a driver supports batch updates, invoke `supportsBatchUpdates()` on a `DatabaseMetaData` instance. If the driver supports batch updates, it will return `true`. The JDBC drivers for MySQL, Access, and Oracle all support batch updates.

To demonstrate batch processing, consider writing a program that gets data from a text file and copies the data from the text file to a table, as shown in Figure 35.2. The text file consists of lines that each corresponds to a row in the table. The fields in a row are separated by commas. The string values in a row are enclosed in single quotes. You can view the text file by clicking the *View File* button and copy the text to the table by clicking the *Copy* button. The table must already be defined in the database. Figure 35.2 shows the text file `table.txt` copied to table `Person`. `Person` is created using the following statement:

```
create table Person (
  firstName varchar(20),
  mi char(1),
  lastName varchar(20)
)
```



Filename c:\book\table.txt

```
'John', 'F', 'Adams'
'Kate', 'R', 'Dawson'
'Cindy', 'J', 'Fox'
'George', 'D', 'Adams'
'Peter', 'S', 'Tse'
```

JDBC Driver jdbc:mysql://localhost/javabook
Database ... com.mysql.jdbc.Driver
Username scott
Password •••••
Table Name Person

Batch updates completed

FIGURE 35.2 The `CopyFileToTable` utility copies text files to database tables.

Listing 35.2 gives the solution to the problem.

LISTING 35.2 CopyFileToTable.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.sql.*;
import java.util.Scanner;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.SplitPane;
```

```java
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CopyFileToTable extends Application {
  // Text file info
  private TextField tfFilename = new TextField();
  private TextArea taFile = new TextArea();

  // JDBC and table info
  private ComboBox<String> cboURL = new ComboBox<>();
  private ComboBox<String> cboDriver = new ComboBox<>();
  private TextField tfUsername = new TextField();
  private PasswordField pfPassword = new PasswordField();
  private TextField tfTableName = new TextField();

  private Button btViewFile = new Button("View File");
  private Button btCopy = new Button("Copy");
  private Label lblStatus = new Label();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    cboURL.getItems().addAll(FXCollections.observableArrayList(
      "jdbc:mysql://localhost/javabook",
      "jdbc:mysql://liang.armstrong.edu/javabook",
      "jdbc:odbc:exampleMDBDataSource",
      "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"));
    cboURL.getSelectionModel().selectFirst();

    cboDriver.getItems().addAll(FXCollections.observableArrayList(
      "com.mysql.jdbc.Driver", "sun.jdbc.odbc.dbcOdbcDriver",
      "oracle.jdbc.driver.OracleDriver"));
    cboDriver.getSelectionModel().selectFirst();

    // Create UI for connecting to the database
    GridPane gridPane = new GridPane();
    gridPane.add(new Label("JDBC Driver"), 0, 0);
    gridPane.add(new Label("Database URL"), 0, 1);
    gridPane.add(new Label("Username"), 0, 2);
    gridPane.add(new Label("Password"), 0, 3);
    gridPane.add(new Label("Table Name"), 0, 4);
    gridPane.add(cboURL, 1, 0);
    gridPane.add(cboDriver, 1, 1);
    gridPane.add(tfUsername, 1, 2);
    gridPane.add(pfPassword, 1, 3);
    gridPane.add(tfTableName, 1, 4);

    HBox hBoxConnection = new HBox(10);
    hBoxConnection.getChildren().addAll(lblStatus, btCopy);
    hBoxConnection.setAlignment(Pos.CENTER_RIGHT);

    VBox vBoxConnection = new VBox(5);
    vBoxConnection.getChildren().addAll(
      new Label("Target Database Table"),
      gridPane, hBoxConnection);

    gridPane.setStyle("-fx-border-color: black;");

    BorderPane borderPaneFileName = new BorderPane();
    borderPaneFileName.setLeft(new Label("Filename"));
    borderPaneFileName.setCenter(tfFilename);
    borderPaneFileName.setRight(btViewFile);

    BorderPane borderPaneFileContent = new BorderPane();
    borderPaneFileContent.setTop(borderPaneFileName);
    borderPaneFileContent.setCenter(taFile);

    BorderPane borderPaneFileSource = new BorderPane();
    borderPaneFileSource.setTop(new Label("Source Text File"));
    borderPaneFileSource.setCenter(borderPaneFileContent);

    SplitPane sp = new SplitPane();
    sp.getItems().addAll(borderPaneFileSource, vBoxConnection);

    // Create a scene and place it in the stage
    Scene scene = new Scene(sp, 680, 230);
    primaryStage.setTitle("CopyFileToTable"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    btViewFile.setOnAction(e -> showFile());
    btCopy.setOnAction(e -> {
      try {
        copyFile();
      }
      catch (Exception ex) {
        lblStatus.setText(ex.toString());
      }
    });
  }

  /** Display the file in the text area */
  private void showFile() {
    Scanner input = null;
    try {
      // Use a Scanner to read text from the file
      input = new Scanner(new File(tfFilename.getText().trim()));

      // Read a line and append the line to the text area
      while (input.hasNext())
        taFile.appendText(input.nextLine() + '\n');
    }
    catch (FileNotFoundException ex) {
      System.out.println("File not found: " + tfFilename.getText());
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
    finally {
      if (input != null) input.close();
    }
  }

  private void copyFile() throws Exception {
    // Load the JDBC driver
    Class.forName(cboDriver.getSelectionModel()
      .getSelectedItem().trim());
    System.out.println("Driver loaded");

    // Establish a connection
    Connection conn = DriverManager.getConnection(
      cboURL.getSelectionModel().getSelectedItem().trim(),
      tfUsername.getText().trim(),
      String.valueOf(pfPassword.getText()).trim());
    System.out.println("Database connected");

    // Read each line from the text file and insert it to the table
    insertRows(conn);
  }

  private void insertRows(Connection connection) {
    // Build the SQL INSERT statement
    String sqlInsert = "insert into " + tfTableName.getText()
      + " values (";

    // Use a Scanner to read text from the file
    Scanner input = null;

    // Get file name from the text field
    String filename = tfFilename.getText().trim();

    try {
      // Create a scanner
      input = new Scanner(new File(filename));

      // Create a statement
      Statement statement = connection.createStatement();

      System.out.println("Driver major version? " +
        connection.getMetaData().getDriverMajorVersion());

      // Determine if batchUpdatesSupported is supported
      boolean batchUpdatesSupported = false;

      try {
        if (connection.getMetaData().supportsBatchUpdates()) {
          batchUpdatesSupported = true;
          System.out.println("batch updates supported");
        }
        else {
          System.out.println("The driver " +
            "does not support batch updates");
        }
      }
      catch (UnsupportedOperationException ex) {
        System.out.println("The operation is not supported");
      }

      // Determine if the driver is capable of batch updates
      if (batchUpdatesSupported) {
        // Read a line and add the insert table command to the batch
        while (input.hasNext()) {
          statement.addBatch(sqlInsert + input.nextLine() + ")");
        }

        statement.executeBatch();

        lblStatus.setText("Batch updates completed");
      }
```

```
198         else {
199           // Read a line and execute insert table command
200           while (input.hasNext()) {
201             statement.executeUpdate(sqlInsert + input.nextLine() + ")");
202           }
203 
204           lblStatus.setText("Single row update completed");
205         }
206       }
207       catch (SQLException ex) {
208         System.out.println(ex);
209       }
210       catch (FileNotFoundException ex) {
211         System.out.println("File not found: " + filename);
212       }
213       finally {
214         if (input != null) input.close();
215       }
216     }
217   }
```

The `insertRows` method (lines 149–216) uses the batch updates to submit SQL INSERT commands to the database for execution, if the driver supports batch updates. Lines 174–181 check whether the driver supports batch updates. If the driver does not support the operation, an `UnsupportedOperationException` exception will be thrown (line 183) when the `supportsBatchUpdates()` method is invoked.

The tables must already be created in the database. The file format and contents must match the database table specification. Otherwise, the SQL INSERT command will fail.

In Exercise 35.1, you will write a program to insert a thousand records to a database and compare the performance with and without batch updates.

Check Point

35.3.1 What is batch processing in JDBC? What are the benefits of using batch processing?

35.3.2 How do you add an SQL statement to a batch? How do you execute a batch?

35.3.3 Can you execute a SELECT statement in a batch?

35.3.4 How do you know whether a JDBC driver supports batch updates?

35.4 Scrollable and Updatable Result Set

Key Point

You can use scrollable and updatable result set to move the cursor anywhere in the result set to perform insertion, deletion, and update.

The result sets used in the preceding examples are read sequentially. A result set maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The `next()` method moves the cursor forward to the next row. This is known as *sequential forward reading*.

A more powerful way of accessing database is to use a scrollable and updatable result, which enables you to scroll the rows both forward and backward and move the cursor to a desired location using the `first`, `last`, `next`, `previous`, `absolute`, or `relative` method. Additionally, you can insert, delete, or update a row in the result set and have the changes automatically reflected in the database.

To obtain a scrollable or updatable result set, you must first create a statement with an appropriate type and concurrency mode. For a static statement, use

```
Statement statement = connection.createStatement
  (int resultSetType, int resultSetConcurrency);
```

For a prepared statement, use

```
PreparedStatement statement = connection.prepareStatement
  (String sql, int resultSetType, int resultSetConcurrency);
```

The possible values of `resultSetType` are the constants defined in the `ResultSet`:

- `TYPE_FORWARD_ONLY`: The result set is accessed forward sequentially.
- `TYPE_SCROLL_INSENSITIVE`: The result set is scrollable, but not sensitive to changes in the database.
- `TYPE_SCROLL_SENSITIVE`: The result set is scrollable and sensitive to changes made by others. Use this type if you want the result set to be scrollable and updatable.

The possible values of `resultSetConcurrency` are the constants defined in the `ResultSet`:

- `CONCUR_READ_ONLY`: The result set cannot be used to update the database.
- `CONCUR_UPDATABLE`: The result set can be used to update the database.

For example, if you want the result set to be scrollable and updatable, you can create a statement, as follows:

```
Statement statement = connection.createStatement
  (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)
```

You use the `executeQuery` method in a `Statement` object to execute an SQL query that returns a result set as follows:

```
ResultSet resultSet = statement.executeQuery(query);
```

You can now use the methods `first()`, `next()`, `previous()`, and `last()` to move the cursor to the first row, next row, previous row, and last row. The `absolute(int row)` method moves the cursor to the specified row; and the `getXxx(int columnIndex)` or `getXxx(String columnName)` method is used to retrieve the value of a specified field at the current row. The methods `insertRow()`, `deleteRow()`, and `updateRow()` can also be used to insert, delete, and update the current row. Before applying `insertRow` or `updateRow`, you need to use the method `updateXxx(int columnIndex, Xxx value)` or `update(String columnName, Xxx value)` to write a new value to the field at the current row. The `cancelRowUpdates()` method cancels the updates made to a row. The `close()` method closes the result set and releases its resource. The `wasNull()` method returns true if the last column read had a value of SQL NULL.

Listing 35.3 gives an example that demonstrates how to create a scrollable and updatable result set. The program creates a result set for the `StateCapital` table. The `StateCapital` table is defined as follows:

```
create table StateCapital (
  state varchar(40),
  capital varchar(40)
);
```

LISTING 35.3 ScrollUpdateResultSet.java

```
import java.sql.*;

public class ScrollUpdateResultSet {
  public static void main(String[] args)
      throws SQLException, ClassNotFoundException {
```

```
    // Load the JDBC driver
    Class.forName("oracle.jdbc.driver.OracleDriver");
    System.out.println("Driver loaded");

    // Connect to a database
    Connection connection = DriverManager.getConnection
      ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
       "scott", "tiger");
    connection.setAutoCommit(true);
    System.out.println("Database connected");

    // Get a new statement for the current connection
    Statement statement = connection.createStatement(
      ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

    // Get ResultSet
    ResultSet resultSet = statement.executeQuery
      ("select state, capital from StateCapital");

    System.out.println("Before update ");
    displayResultSet(resultSet);

    // Update the second row
    resultSet.absolute(2); // Move cursor to the second row
    resultSet.updateString("state", "New S"); // Update the column
    resultSet.updateString("capital", "New C"); // Update the column
    resultSet.updateRow(); // Update the row in the data source

    // Insert after the last row
    resultSet.last();
    resultSet.moveToInsertRow(); // Move cursor to the insert row
    resultSet.updateString("state", "Florida");
    resultSet.updateString("capital", "Tallahassee");
    resultSet.insertRow(); // Insert the row
    resultSet.moveToCurrentRow(); // Move the cursor to the current row

    // Delete fourth row
    resultSet.absolute(4); // Move cursor to the 5th row
    resultSet.deleteRow(); // Delete the second row

    System.out.println("After update ");
    resultSet = statement.executeQuery
      ("select state, capital from StateCapital");
    displayResultSet(resultSet);

    // Close the connection
    resultSet.close();
  }

  private static void displayResultSet(ResultSet resultSet)
      throws SQLException {
    ResultSetMetaData rsMetaData = resultSet.getMetaData();
    resultSet.beforeFirst();
    while (resultSet.next()) {
      for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
        System.out.printf("%-12s\t", resultSet.getObject(i));
      System.out.println();
    }
  }
}
```

```
Driver loaded
Database connected

Before update
Indiana         Indianapolis
Illinois        Springfield
California      Sacramento
Georgia         Atlanta
Texas           Austin

After update
Indiana         Indianapolis
New S           New C
California      Sacramento
Texas           Austin
Florida         Tallahassee
```

The code in lines 18–19 creates a `Statement` for producing scrollable and updatable result sets.

The program moves the cursor to the second row in the result set (line 29), updates two columns in this row (lines 30–31), and invokes the `updateRow()` method to update the row in the underlying database (line 32).

An updatable `ResultSet` object has a special row associated with it that serves as a staging area for building a row to be inserted. This special row is called the *insert row*. To insert a row, first invoke the `moveToInsertRow()` method to move the cursor to the insert row (line 36), then update the columns using the `updateXxx` method (lines 37–38), and finally insert the row using the `insertRow()` method (line 39). Invoking `moveToCurrentRow()` moves the cursor to the current inserted row (lines 40).

The program moves to the fourth row and invokes the `deleteRow()` method to delete the row from the database (lines 43–44).

Note

Not all current drivers support scrollable and updatable result sets. The example is tested using Oracle ojdbc6 driver. You can use `supportsResultSetType(int type)` and `supportsResultSetConcurrency(int type, int concurrency)` in the `DatabaseMetaData` interface to find out which result type and currency modes are supported by the JDBC driver. But even if a driver supports the scrollable and updatable result set, a result set for a complex query might not be able to perform an update. For example, the result set for a query that involves several tables is likely not to support update operations.

Note

The program may not work due to an issue in the Oracle JDBC driver if lines 22–23 are replaced by

```
ResultSet resultSet = statement.executeQuery
  ("select * from StateCapital");
```

Check Point

35.4.1 What is a scrollable result set? What is an updatable result set?

35.4.2 How do you create a scrollable and updatable `ResultSet`?

35.4.3 How do you know whether a JDBC driver supports a scrollable and updatable `ResultSet`?

35.5 RowSet, JdbcRowSet, and CachedRowSet

The RowSet interface can be used to simplify database programming.

Key Point

The `RowSet` interface extends `java.sql.ResultSet` with additional capabilities that allow a `RowSet` instance to be configured to connect to a JDBC url, username, and password, set an SQL command, execute the command, and retrieve the execution result. In essence, it combines `Connection`, `Statement`, and `ResultSet` into one interface.

Note

Not all JDBC drivers support `RowSet`. Currently, the JDBC-ODBC driver does not support all features of `RowSet`.

35.5.1 RowSet Basics

There are two types of `RowSet` objects: connected and disconnected. A *connected* `RowSet` object makes a connection with a data source and maintains that connection throughout its life cycle. A disconnected `RowSet` object makes a connection with a data source, executes a query to get data from the data source, and then closes the connection. A *disconnected* rowset may make changes to its data while it is disconnected and then send the changes back to the original source of the data, but it must reestablish a connection to do so.

There are several versions of `RowSet`. Two frequently used are `JdbcRowSet` and `CachedRowSet`. Both are subinterfaces of `RowSet`. `JdbcRowSet` is connected, while `CachedRowSet` is disconnected. Also, `JdbcRowSet` is neither serializable nor cloneable, while `CachedRowSet` is both. The database vendors are free to provide concrete implementations for these interfaces. Oracle has provided the reference implementation `JdbcRowSetImpl` for `JdbcRowSet` and `CachedRowSetImpl` for `CachedRowSet`. Figure 35.3 shows the relationship of these components.

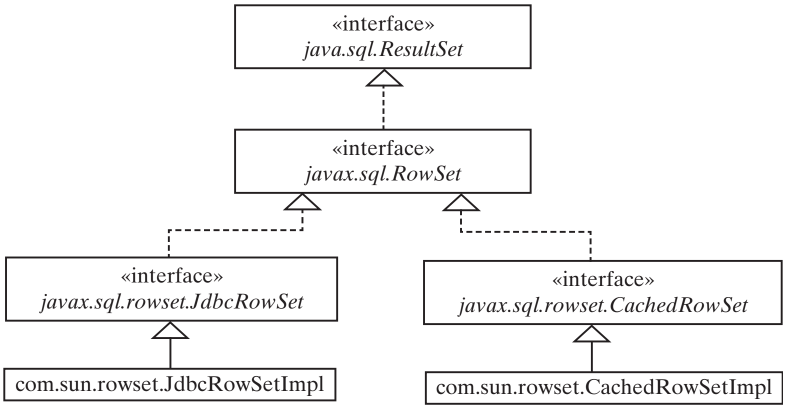

«interface»
java.sql.ResultSet
«interface»
javax.sql.RowSet
«interface»
javax.sql.rowset.JdbcRowSet
«interface»
javax.sql.rowset.CachedRowSet
com.sun.rowset.JdbcRowSetImpl
com.sun.rowset.CachedRowSetImpl

Figure 35.3 The `JdbcRowSetImpl` and `CachedRowSetImpl` are concrete implementations of `RowSet`.

The `RowSet` interface contains the JavaBeans properties with getter and setter methods. You can use the setter methods to set a new url, username, password, and command for an SQL statement. Using a `RowSet`, Listing 34.1 can be simplified, as shown in Listing 35.4.

Listing 35.4 `SimpleRowSet.java`

```
import java.sql.SQLException;
import javax.sql.RowSet;
import com.sun.rowset.*;

```

```
 5  public class SimpleRowSet {
 6    public static void main(String[] args)
 7        throws SQLException, ClassNotFoundException {
 8      // Load the JDBC driver
 9      Class.forName("com.mysql.jdbc.Driver");
10      System.out.println("Driver loaded");
11
12      // Create a row set
13      RowSet rowSet = new JdbcRowSetImpl();
14
15      // Set RowSet properties
16      rowSet.setUrl("jdbc:mysql://localhost/javabook");
17      rowSet.setUsername("scott");
18      rowSet.setPassword("tiger");
19      rowSet.setCommand("select firstName, mi, lastName " +
20        "from Student where lastName = 'Smith'");
21      rowSet.execute();
22
23      // Iterate through the result and print the student names
24      while (rowSet.next())
25        System.out.println(rowSet.getString(1) + "\t" +
26          rowSet.getString(2) + "\t" + rowSet.getString(3));
27
28      // Close the connection
29      rowSet.close();
30    }
31  }
```

Line 13 creates a `RowSet` object using `JdbcRowSetImpl`. The program uses the `RowSet`'s set method to set a URL, username, and password (lines 16–18) and a command for a query statement (line 19). Line 24 executes the command in the `RowSet`. The methods `next()` and `getString(int)` for processing the query result (lines 25–26) are inherited from `ResultSet`.

If you replace `JdbcRowSet` with `CachedRowSet` in line 13, the program will work just fine. Note, the JDBC-ODBC driver supports `JdbcRowSetImpl`, but not `CachedRowSetImpl`.

Tip

Since `RowSet` is a subinterface of `ResultSet`, all the methods in `ResultSet` can be used in `RowSet`. For example, you can obtain `ResultSetMetaData` from a `RowSet` using the `getMetaData()` method.

35.5.2 RowSet for PreparedStatement

The discussion in §34.5, "PreparedStatement," introduced processing parameterized SQL statements using the `PreparedStatement` interface. `RowSet` has the capability to support parameterized SQL statements. The set methods for setting parameter values in `PreparedStatement` are implemented in `RowSet`. You can use these methods to set parameter values for a parameterized SQL command. Listing 35.5 demonstrates how to use a parameterized statement in `RowSet`. Line 19 sets an SQL query statement with two parameters for `lastName` and `mi` in a `RowSet`. Since these two parameters are strings, the `setString` method is used to set actual values in lines 21–22.

LISTING 35.5 `RowSetPreparedStatement.java`

```
1  import java.sql.*;
2  import javax.sql.RowSet;
3  import com.sun.rowset.*;
4
```

```
public class RowSetPreparedStatement {
 public static void main(String[] args)
   throws SQLException, ClassNotFoundException {
  // Load the JDBC driver
  Class.forName("com.mysql.jdbc.Driver");
  System.out.println("Driver loaded");

  // Create a row set
  RowSet rowSet = new JdbcRowSetImpl();

  // Set RowSet properties
  rowSet.setUrl("jdbc:mysql://localhost/javabook");
  rowSet.setUsername("scott");
  rowSet.setPassword("tiger");
  rowSet.setCommand("select * from Student where lastName = ? " +
   "and mi = ?");
  rowSet.setString(1, "Smith");
  rowSet.setString(2, "R");
  rowSet.execute();

  ResultSetMetaData rsMetaData = rowSet.getMetaData();
  for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
   System.out.printf("%-12s\t", rsMetaData.getColumnName(i));
  System.out.println();

  // Iterate through the result and print the student names
  while (rowSet.next()) {
   for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
    System.out.printf("%-12s\t", rowSet.getObject(i));
   System.out.println();
  }

  // Close the connection
  rowSet.close();
 }
}
```

35.5.3 Scrolling and Updating RowSet

By default, a **`ResultSet`** object is neither scrollable nor updatable. However, a **`RowSet`** object is both. It is easier to scroll and update a database through a **`RowSet`** than through a **`ResultSet`**. Listing 35.6 rewrites Listing 35.3 using a **`RowSet`**. You can use methods such as **`absolute(int)`** to move the cursor and methods such as **`delete()`**, **`updateRow()`**, and **`insertRow()`** to update the database.

LISTING 35.6 ScrollUpdateRowSet.java

```
import java.sql.*;
import javax.sql.RowSet;
import com.sun.rowset.JdbcRowSetImpl;

public class ScrollUpdateRowSet {
 public static void main(String[] args)
   throws SQLException, ClassNotFoundException {
  // Load the JDBC driver
  Class.forName("com.mysql.jdbc.Driver");
  System.out.println("Driver loaded");

12      // Create a row set
13      RowSet rowSet = new JdbcRowSetImpl();
14
15      // Set RowSet properties
16      rowSet.setUrl("jdbc:mysql://localhost/javabook");
17      rowSet.setUsername("scott");
18      rowSet.setPassword("tiger");
19      rowSet.setCommand("select state, capital from StateCapital");
20      rowSet.execute();
21
22      System.out.println("Before update ");
23      displayRowSet(rowSet);
24
25      // Update the second row
26      rowSet.absolute(2); // Move cursor to the 2nd row
27      rowSet.updateString("state", "New S"); // Update the column
28      rowSet.updateString("capital", "New C"); // Update the column
29      rowSet.updateRow(); // Update the row in the data source
30
31      // Insert after the second row
32      rowSet.last();
33      rowSet.moveToInsertRow(); // Move cursor to the insert row
34      rowSet.updateString("state", "Florida");
35      rowSet.updateString("capital", "Tallahassee");
36      rowSet.insertRow(); // Insert the row
37      rowSet.moveToCurrentRow(); // Move the cursor to the current row
38
39      // Delete fourth row
40      rowSet.absolute(4); // Move cursor to the fifth row
41      rowSet.deleteRow(); // Delete the second row
42
43      System.out.println("After update ");
44      displayRowSet(rowSet);
45
46      // Close the connection
47      rowSet.close();
48    }
49
50    private static void displayRowSet(RowSet rowSet)
51        throws SQLException {
52      ResultSetMetaData rsMetaData = rowSet.getMetaData();
53      rowSet.beforeFirst();
54      while (rowSet.next()) {
55        for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
56          System.out.printf("%-12s\t", rowSet.getObject(i));
57        System.out.println();
58      }
59    }
60  }
```

If you replace `JdbcRowSet` with `CachedRowSet` in line 13, the database is not changed. To make the changes on the `CachedRowSet` effective in the database, you must invoke the `acceptChanges()` method after you make all the changes, as follows:

```
// Write changes back to the database
((com.sun.rowset.CachedRowSetImpl)rowSet).acceptChanges();
```

This method automatically reconnects to the database and writes all the changes back to the database.

35.5.4 RowSetEvent

A `RowSet` object fires a `RowSetEvent` whenever the object's cursor has moved, a row has changed, or the entire row set has changed. This event can be used to synchronize a `RowSet` with the components that rely on the `RowSet`. For example, a visual component that displays the contents of a `RowSet` should be synchronized with the `RowSet`. The `RowSetEvent` can be used to achieve synchronization. The handlers in `RowSetListener` are `cursorMoved(RowSetEvent)`, `rowChanged(RowSetEvent)`, and `cursorSetChanged(RowSetEvent)`.

Listing 35.7 gives an example that demonstrates `RowSetEvent`. A listener for `RowSetEvent` is registered in lines 14–26. When `rowSet.execute()` (line 33) is executed, the entire row set is changed, so the listener's `rowSetChanged` handler is invoked. When `rowSet.last()` (line 35) is executed, the cursor is moved, so the listener's `cursorMoved` handler is invoked. When `rowSet.updateRow()` (line 37) is executed, the row is updated, so the listener's `rowChanged` handler is invoked.

LISTING 35.7 `TestRowSetEvent.java`

```
 1  import java.sql.*;
 2  import javax.sql.*;
 3  import com.sun.rowset.*;
 4
 5  public class TestRowSetEvent {
 6    public static void main(String[] args)
 7        throws SQLException, ClassNotFoundException {
 8      // Load the JDBC driver
 9      Class.forName("com.mysql.jdbc.Driver");
10      System.out.println("Driver loaded");
11
12      // Create a row set
13      RowSet rowSet = new JdbcRowSetImpl();
14      rowSet.addRowSetListener(new RowSetListener() {
15        public void cursorMoved(RowSetEvent e) {
16          System.out.println("Cursor moved");
17        }
18
19        public void rowChanged(RowSetEvent e) {
20          System.out.println("Row changed");
21        }
22
23        public void rowSetChanged(RowSetEvent e) {
24          System.out.println("row set changed");
25        }
26      });
27
28      // Set RowSet properties
29      rowSet.setUrl("jdbc:mysql://localhost/javabook");
30      rowSet.setUsername("scott");
31      rowSet.setPassword("tiger");
32      rowSet.setCommand("select * from Student");
33      rowSet.execute();
34
35      rowSet.last(); // Cursor moved
36      rowSet.updateString("lastName", "Yao"); // Update column
37      rowSet.updateRow(); // Row updated
38
39      // Close the connection
40      rowSet.close();
41    }
42  }
```

Check Point

35.5.1 What are the advantages of `RowSet`?

35.5.2 What are `JdbcRowSet` and `CachedRowSet`? What are the differences between them?

35.5.3 How do you create a `JdbcRowSet` and a `CachedRowSet`?

35.5.4 Can you scroll and update a `RowSet`? What method must be invoked to write the changes in a `CachedRowSet` to the database?

35.5.5 Describe the handlers in `RowSetListener`.

35.6 Storing and Retrieving Images in JDBC

Key Point

You can store and retrieve images using JDBC.

A database can store not only numbers and strings, but also images. SQL3 introduced a new data type called BLOB (*B*inary *L*arge *OB*ject) for storing binary data, which can be used to store images. Another new SQL3 type is CLOB (*C*haracter *L*arge *OB*ject) for storing a large text in the character format. JDBC introduced the interfaces `java.sql.Blob` and `java.sql.Clob` to support mapping for these new SQL types. You can use `getBlob`, `setBinaryStream`, `getClob`, `setBlob`, and `setClob`, to access SQL BLOB and CLOB values in the interfaces `ResultSet` and `PreparedStatement`.

To store an image into a cell in a table, the corresponding column for the cell must be of the BLOB type. For example, the following SQL statement creates a table whose type for the flag column is BLOB:

```
create table Country(name varchar(30), flag blob,
  description varchar(255));
```

In the preceding statement, the `description` column is limited to 255 characters, which is the upper limit for MySQL. For Oracle, the upper limit is 32,672 bytes. For a large character field, you can use the CLOB type for Oracle, which can store up to two GB of characters. MySQL does not support CLOB. However, you can use BLOB to store a long string and convert binary data into characters.

Note

MS Access database does not support the BLOB and CLOB types.

To insert a record with images to a table, define a prepared statement like this one:

```
  PreparedStatement pstmt = connection.prepareStatement(
   "insert into Country values(?, ?, ?)");
```

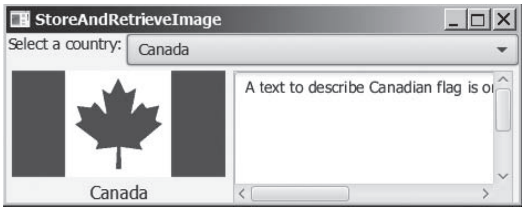
Images are usually stored in files. You may first get an instance of `InputStream` for an image file then use the `setBinaryStream` method to associate the input stream with a cell in the table, as follows:

```
// Store image to the table cell
File file = new File(imageFilename);
InputStream inputImage = new FileInputStream(file);
pstmt.setBinaryStream(2, inputImage, (int)(file.length()));
```

To retrieve an image from a table, use the `getBlob` method, as shown below:

```
// Store image to the table cell
Blob blob = rs.getBlob(1);
ImageIcon imageIcon = new ImageIcon(
  blob.getBytes(1, (int)blob.length()));
```

Listing 35.8 gives a program that demonstrates how to store and retrieve images in JDBC. The program first creates the `Country` table and stores data to it. Then the program retrieves the country names from the table and adds them to a combo box. When the user selects a name from the combo box, the country's flag and description are displayed, as shown in Figure 35.4.



Canada

A text to describe Canadian flag is o

FIGURE 35.4 The program enables you to retrieve data, including images, from a table and displays them.

LISTING 35.8 `StoreAndRetrieveImage.java`

```java
import java.sql.*;
import java.io.*;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class StoreAndRetrieveImage extends Application {
  // Connection to the database
  private Connection connection;

  // Statement for static SQL statements
  private Statement stmt;

  // Prepared statement
  private PreparedStatement pstmt = null;
  private DescriptionPane descriptionPane
    = new DescriptionPane();

  private ComboBox<String> cboCountry = new ComboBox<>();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    try {
      connectDB(); // Connect to DB
      storeDataToTable(); //Store data to the table (including image)
      fillDataInComboBox(); // Fill in combo box
      retrieveFlagInfo(cboCountry.getSelectionModel().getSelectedItem());
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }

    BorderPane paneForComboBox = new BorderPane();
    paneForComboBox.setLeft(new Label("Select a country: "));
    paneForComboBox.setCenter(cboCountry);
    cboCountry.setPrefWidth(400);
    BorderPane pane = new BorderPane();
    pane.setTop(paneForComboBox);
    pane.setCenter(descriptionPane);

    Scene scene = new Scene(pane, 350, 150);
    primaryStage.setTitle("StoreAndRetrieveImage");
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    cboCountry.setOnAction(e ->
      retrieveFlagInfo(cboCountry.getValue()));
  }

  private void connectDB() throws Exception {
    // Load the driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

    // Establish connection
    connection = DriverManager.getConnection
      ("jdbc:mysql://localhost/javabook", "scott", "tiger");
    System.out.println("Database connected");

    // Create a statement for static SQL
    stmt = connection.createStatement();

    // Create a prepared statement to retrieve flag and description
    pstmt = connection.prepareStatement("select flag, description " +
      "from Country where name = ?");
  }

  private void storeDataToTable() {
    String[] countries = {"Canada", "UK", "USA", "Germany",
      "Indian", "China"};

   String[] imageFilenames = {"image/ca.gif", "image/uk.gif",
      "image/us.gif", "image/germany.gif", "image/india.gif",
      "image/china.gif"};

   String[] descriptions = {"A text to describe Canadian " +
      "flag is omitted", "British flag ...", "American flag ...",
      "German flag ...", "Indian flag ...", "Chinese flag ..."};

    try {
      // Create a prepared statement to insert records
      PreparedStatement pstmt = connection.prepareStatement(
        "insert into Country values(?, ?, ?)");

      // Store all predefined records
      for (int i = 0; i < countries.length; i++) {
        pstmt.setString(1, countries[i]);

        // Store image to the table cell
        java.net.URL url =
          this.getClass().getResource(imageFilenames[i]);
        InputStream inputImage = url.openStream();
        pstmt.setBinaryStream(2, inputImage,
          (int)(inputImage.available()));

        pstmt.setString(3, descriptions[i]);
        pstmt.executeUpdate();
```

```
103         }
104 
105         System.out.println("Table Country populated");
106       }
107       catch (Exception ex) {
108         ex.printStackTrace();
109       }
110     }
111 
112     private void fillDataInComboBox()  throws Exception {
113       ResultSet rs = stmt.executeQuery("select name from Country");
114       while (rs.next()) {
115         cboCountry.getItems().add(rs.getString(1));
116       }
117       cboCountry.getSelectionModel().selectFirst();
118     }
119 
120     private void retrieveFlagInfo(String name) {
121       try {
122         pstmt.setString(1, name);
123         ResultSet rs = pstmt.executeQuery();
124         if (rs.next()) {
125           Blob blob = rs.getBlob(1);
126           ByteArrayInputStream in = new ByteArrayInputStream
127             (blob.getBytes(1, (int)blob.length()));
128           Image image = new Image(in);
129           ImageView imageView = new ImageView(image);
130           descriptionPane.setImageView(imageView);
131           descriptionPane.setTitle(name);
132           String description = rs.getString(2);
133           descriptionPane.setDescription(description);
134         }
135       }
136       catch (Exception ex) {
137         System.err.println(ex);
138       }
139     }
140   }
```

`DescriptionPane` (line 21) is a component for displaying a country (name, flag, and description). This component was presented in Listing 16.6, DescriptionPane.java.

The `storeDataToTable` method (lines 73–110) populates the table with data. The `fillDataInComboBox` method (lines 112–118) retrieves the country names and adds them to the combo box. The `retrieveFlagInfo(name)` method (lines 120–139) retrieves the flag and description for the specified country name.

Check Point

35.6.1 How do you store images into a database?

35.6.2 How do you retrieve images from a database?

35.6.3 Does Oracle support the SQL3 BLOB type and CLOB type? What about MySQL and Access?

KEY TERMS

BLOB type	35-20
CLOB type	35-20
batch mode	35-2
cached row set	35-15
row set	35-15
scrollable result set	35-2
updatable result set	35-11

Chapter Summary

1. This chapter developed a universal SQL client that can be used to access any local or remote relational database.

2. You can use the `addBatch(SQLString)` method to add SQL statements to a statement for batch processing.

3. You can create a statement to specify that the result set be scrollable and updatable. By default, the result set is neither of these.

4. The `RowSet` can be used to simplify Java database programming. A `RowSet` object is scrollable and updatable. A `RowSet` can fire a `RowSetEvent`.

5. You can store and retrieve image data in JDBC using the SQL BLOB type.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

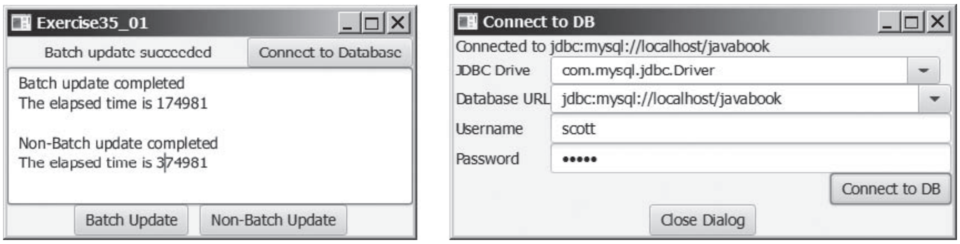
MyProgrammingLab™

Programming Exercises

***35.1** (*Batch update*) Write a program that inserts a thousand records to a database, and compare the performance with and without batch updates, as shown in Figure 35.5a. Suppose the table is defined as follows:

```
create table Temp(num1 double, num2 double, num3 double)
```

Use the `Math.random()` method to generate random numbers for each record. Create a dialog box that contains `DBConnectionPanel`, discussed in Exercise 34.3. Use this dialog box to connect to the database. When you click the *Connect to Database* button in Figure 35.5a, the dialog box in Figure 35.5b is displayed.



```
Batch update completed
The elapsed time is 174981

Non-Batch update completed
The elapsed time is 374981
```

Connected to jdbc:mysql://localhost/javabook

JDBC Drive: com.mysql.jdbc.Driver

Database URL: jdbc:mysql://localhost/javabook

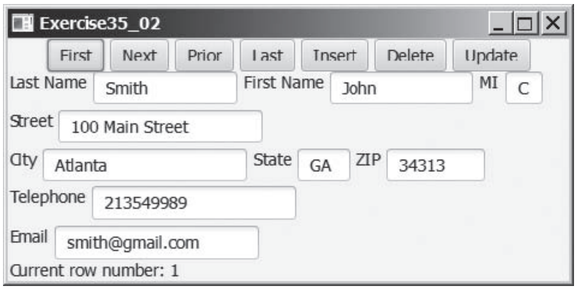
Username: scott

Password: •••••

(a) (b)

Figure 35.5 The program demonstrates the performance improvements that result from using batch updates.

****35.2** (*Scrollable result set*) Write a program that uses the buttons *First*, *Next*, *Prior*, *Last*, *Insert*, *Delete*, and *Update*, and modify a single record in the `Address` table, as shown in Figure 35.6.



Last Name: Smith
First Name: John
MI: C
Street: 100 Main Street
City: Atlanta
State: GA
ZIP: 34313
Telephone: 213549989
Email: smith@gmail.com
Current row number: 1

FIGURE 35.6 You can use the buttons to display and modify a single record in the `Address` table.

The `Address` table is defined as follows:

```
create table Address (
  firstname varchar(25),
  mi char(1),
  lastname varchar(25),
  street varchar(40),
  city varchar(20),
  state varchar(2),
  zip varchar(5),
  telephone varchar(10),
  email varchar(30),
  primary key (firstname, mi, lastname)
);
```

***35.3** (*Display table contents*) Write a program that displays the content for a given table. As shown in Figure 35.7a, you enter a table and click the *Show Contents* button to display the table contents in a table view.

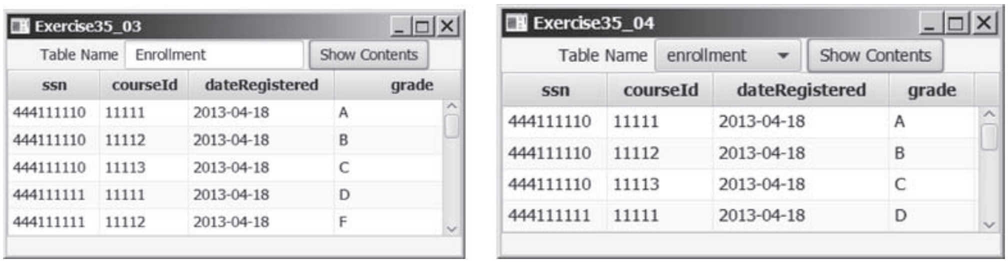


Table Name: Enrollment

ssn	courseId	dateRegistered	grade
444111110	11111	2013-04-18	A
444111110	11112	2013-04-18	B
444111110	11113	2013-04-18	C
444111111	11111	2013-04-18	D
444111111	11112	2013-04-18	F

(a)

Table Name: enrollment

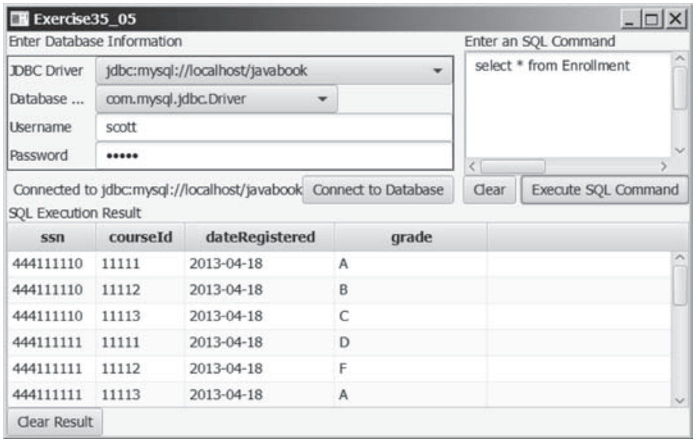
ssn	courseId	dateRegistered	grade
444111110	11111	2013-04-18	A
444111110	11112	2013-04-18	B
444111110	11113	2013-04-18	C
444111111	11111	2013-04-18	D

(b)

FIGURE 35.7 (a) Enter a table name to display the table contents. (b) Select a table name from the combo box to display its contents.

***35.4** (*Find tables and showing their contents*) Write a program that fills in table names in a combo box, as shown in Figure 35.7b. You can select a table from the combo box to display its contents in a table view.

****35.5** (*Revise SQLClient.java*) Rewrite Listing 35.1, SQLClient.java, to display the query result in a `TableView`, as shown in Figure 35.8.


Exercise35_05
Enter Database Information
Enter an SQL Command
Connect to Database
Clear
Execute SQL Command
SQL Execution Result
Clear Result

JDBC Driver: jdbc:mysql://localhost/javabook

Database ...: com.mysql.jdbc.Driver

Username: scott

Password: •••••

select * from Enrollment

Connected to jdbc:mysql://localhost/javabook

ssn	courseId	dateRegistered	grade
444111110	11111	2013-04-18	A
444111110	11112	2013-04-18	B
444111110	11113	2013-04-18	C
444111111	11111	2013-04-18	D
444111111	11112	2013-04-18	F
444111111	11113	2013-04-18	A

FIGURE 35.8 The query result is displayed in a `TableView`.

***35.5** (*Populate Salary table)* Rewrite Programming Exercise 34.8 using a batch mode to improve performance.

CHAPTER 36

INTERNATIONALIZATION

Objectives

- To describe Java's internationalization features (§36.1).
- To construct a locale with language, country, and variant (§36.2).
- To display date and time based on locale (§36.3).
- To display numbers, currencies, and percentages based on locale (§36.4).
- To develop applications for international audiences using resource bundles (§36.5).
- To specify encoding schemes for text I/O (§36.6).

36.1 Introduction

Key Point

This chapter introduces writing Java code for international audience.

Many websites maintain several versions of webpages so that readers can choose one written in a language they understand. Because there are so many languages in the world, it would be highly problematic to create and maintain enough different versions to meet the needs of all clients everywhere. Java comes to the rescue. Java is the first language designed from the ground up to support internationalization. In consequence, it allows your programs to be customized for any number of countries or languages without requiring cumbersome changes in the code.

Here are the major Java features that support internationalization:

- Java characters use *Unicode*, a 16-bit encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world's diverse languages. The use of Unicode encoding makes it easy to write Java programs that can manipulate strings in any international language. (To see all the Unicode characters, visit mindprod.com/jgloss/reuters.html.)
- Java provides the `Locale` class to encapsulate information about a specific locale. A `Locale` object determines how locale-sensitive information, such as date, time, and number, is displayed, and how locale-sensitive operations, such as sorting strings, are performed. The classes for formatting date, time, and numbers, and for sorting strings are grouped in the `java.text` package.
- Java uses the `ResourceBundle` class to separate locale-specific information, such as status messages and GUI component labels, from the program. The information is stored outside the source code and can be accessed and loaded dynamically at runtime from a `ResourceBundle`, rather than hard-coded into the program.

In this chapter, you will learn how to format dates, numbers, currencies, and percentages for different regions, countries, and languages. You will also learn how to use resource bundles to define which images and strings are used by a component, depending on the user's locale and preferences.

36.2 The Locale Class

Key Point

The `Locale` class defines a locale: language and nation.

A `Locale` object represents a geographical, political, or cultural region in which a specific language or custom is used. For example, Americans speak English, and the Chinese speak Chinese. The conventions for formatting dates, numbers, currencies, and percentages may differ from one country to another. The Chinese, for instance, use year/month/day to represent the date, while Americans use month/day/year. It is important to realize that locale is not defined only by country. For example, Canadians speak either Canadian English or Canadian French, depending on which region of Canada they reside in.

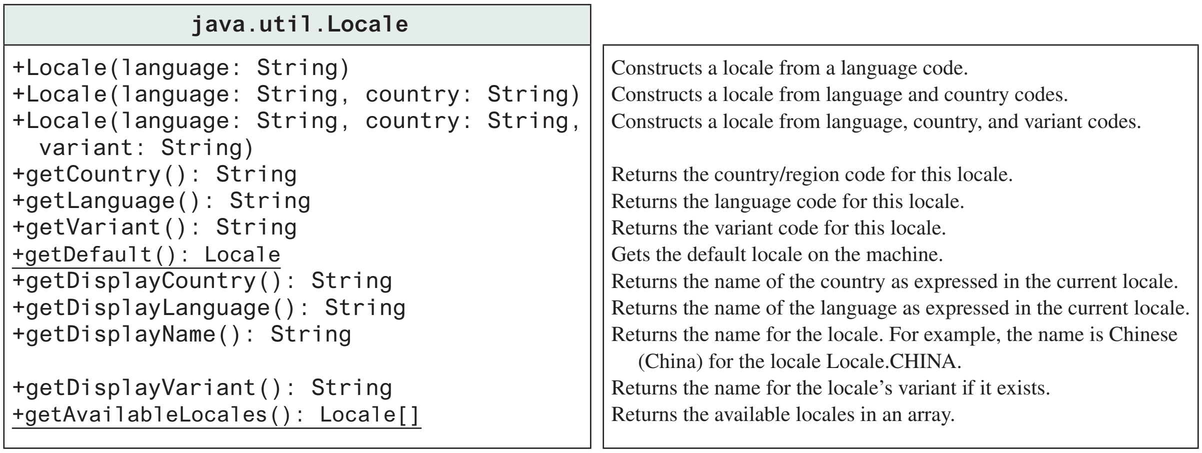
To create a `Locale` object, use one of the three constructors with a specified language and optional country and variant, as shown in Figure 36.1.

The `language` should be a valid language code—that is, one of the lowercase two-letter codes defined by ISO-639. For example, `zh` stands for Chinese, `da` for Danish, `en` for English, `de` for German, and `ko` for Korean. Table 36.1 lists the language codes.

The country should be a valid ISO country code—that is, one of the uppercase, two-letter codes defined by ISO-3166. For example, `CA` stands for Canada, `CN` for China, `DK` for Denmark, `DE` for Germany, and `US` for the United States. Table 36.2 lists the country codes.

The argument variant is rarely used and is needed only for exceptional or system-dependent situations to designate information specific to a browser or vendor. For example, the Norwegian language has two sets of spelling rules, a traditional one called *bokmål* and a new one called *nynorsk*. The locale for traditional spelling would be created as follows:

```
new Locale("no", "NO", "B");
```



java.util.Locale	
`+Locale(language: String)`	Constructs a locale from a language code.
`+Locale(language: String, country: String)`	Constructs a locale from language and country codes.
`+Locale(language: String, country: String, variant: String)`	Constructs a locale from language, country, and variant codes.
`+getCountry(): String`	Returns the country/region code for this locale.
`+getLanguage(): String`	Returns the language code for this locale.
`+getVariant(): String`	Returns the variant code for this locale.
`+getDefault(): Locale`	Gets the default locale on the machine.
`+getDisplayCountry(): String`	Returns the name of the country as expressed in the current locale.
`+getDisplayLanguage(): String`	Returns the name of the language as expressed in the current locale.
`+getDisplayName(): String`	Returns the name for the locale. For example, the name is Chinese (China) for the locale Locale.CHINA.
`+getDisplayVariant(): String`	Returns the name for the locale's variant if it exists.
`+getAvailableLocales(): Locale[]`	Returns the available locales in an array.

FIGURE 36.1 The `Locale` class encapsulates a locale.

TABLE 31.1 Common Language Codes

Code	*Language*	*Code*	*Language*
da	Danish	ja	Japanese
de	German	ko	Korean
el	Greek	nl	Dutch
en	English	no	Norwegian
es	Spanish	pt	Portuguese
fi	Finnish	sv	Swedish
fr	French	tr	Turkish
it	Italian	zh	Chinese

TABLE 31.2 Common Country Codes

Code	*Country*	*Code*	*Country*
AT	Austria	IE	Ireland
BE	Belgium	HK	Hong Kong
CA	Canada	IT	Italy
CH	Switzerland	JP	Japan
CN	China	KR	Korea
DE	Germany	NL	Netherlands
DK	Denmark	NO	Norway
ES	Spain	PT	Portugal
FI	Finland	SE	Sweden
FR	France	TR	Turkey
GB	United Kingdom	TW	Taiwan
GR	Greece	US	United States

For convenience, the `Locale` class contains many predefined locale constants. `Locale.CANADA` is for the country Canada and language English; `Locale.CANADA_FRENCH` is for the country Canada and language French. Several other common constants are:

`Locale.US`, `Locale.UK`, `Locale.FRANCE`, `Locale.GERMANY`, `Locale.ITALY`, `Locale.CHINA`, `Locale.KOREA`, `Locale.JAPAN`, and `Locale.TAIWAN`

The `Locale` class also provides the following constants based on language:

```
Locale.CHINESE, Locale.ENGLISH, Locale.FRENCH, Locale.GERMAN,
Locale.ITALIAN, Locale.JAPANESE, Locale.KOREAN,
Locale.SIMPLIFIED_CHINESE, and Locale.TRADITIONAL_CHINESE
```

Tip

You can invoke the static method `getAvailableLocales()` in the `Locale` class to obtain all the available locales supported in the system. For example,

```
Locale[] availableLocales = Calendar.getAvailableLocales();
```

returns all the locales in an array.

Tip

Your machine has a default locale. You may override it by supplying the language and region parameters when you run the program, as follows:

```
java -Duser.language=zh -Duser.region=CN MainClass
```

An operation that requires a `Locale` to perform its task is called *locale sensitive*. Displaying a number such as a date or time, for example, is a locale-sensitive operation; the number should be formatted according to the customs and conventions of the user's locale. The sections that follow introduce locale-sensitive operations.

Check Point

36.2.1 How does Java support international characters in languages like Chinese and Arabic?

36.2.2 How do you construct a `Locale` object? How do you get all the available locales from a `Calendar` object?

36.2.3 How do you create a locale for the French-speaking region of Canada? How do you create a locale for the Netherlands?

36.3 Displaying Date and Time

Key Point

The representation of date and time is dependent on locale.

Applications often need to obtain date and time. Java provides a system-independent encapsulation of date and time in the `java.util.Date` class; it also provides `java.util.TimeZone` for dealing with time zones, and `java.util.Calendar` for extracting detailed information from `Date`. Different locales have different conventions for displaying date and time. Should the year, month, or day be displayed first? Should slashes, periods, or colons be used to separate fields of the date? What are the names of the months in the language? The `java.text.DateFormat` class can be used to format date and time in a locale-sensitive way for display to the user. The `Date` class was introduced in Section 9.6.1, "The `Date` Class," and the `Calendar` class and its subclass `GregorianCalendar` were introduced in Section 13.4, "Case Study: `Calendar` and `GregorianCalendar`."

36.3.1 The `TimeZone` Class

`TimeZone` represents a time zone offset and also figures out daylight savings. To get a `TimeZone` object for a specified time zone ID, use `TimeZone.getTimeZone(id)`. To set a time zone in a `Calendar` object, use the `setTimeZone` method with a time zone ID. For example, `cal.setTimeZone(TimeZone.getTimeZone("CST"))` sets the time zone to Central Standard Time. To find all the available time zones supported in Java, use the static method `getAvailableIDs()` in the `TimeZone` class. In general, the international time zone ID is a string in the form of continent/city like Europe/Berlin, Asia/Taipei, and America/Washington. You can also use the static method `getDefault()` in the `TimeZone` class to obtain the default time zone on the host machine.

36.3.2 The `DateFormat` Class

The `DateFormat` class can be used to format date and time in a number of styles. The `DateFormat` class supports several standard formatting styles. To format date and time, simply create an instance of `DateFormat` using one of the three static methods `getDateInstance`, `getTimeInstance`, and `getDateTimeInstance` and apply the `format(Date)` method on the instance, as shown in Figure 36.2.

`java.text.DateFormat`	
`+format(date: Date): String`	Formats a date into a date/time string.
`+getDateInstance(): DateFormat`	Gets the date formatter with the default formatting style for the default locale.
`+getDateInstance(dateStyle: int): DateFormat`	Gets the date formatter with the given formatting style for the default locale.
`+getDateInstance(dateStyle: int, aLocale: Locale): DateFormat`	Gets the date formatter with the given formatting style for the given locale.
`+getDateTimeInstance(): DateFormat`	Gets the date and time formatter with the default formatting style for the default locale.
`+getDateTimeInstance(dateStyle: int, timeStyle: int): DateFormat`	Gets the date and time formatter with the given date and time formatting styles for the default locale.
`+getDateTimeInstance(dateStyle: int, timeStyle: int, aLocale: Locale): DateFormat`	Gets the date and time formatter with the given formatting styles for the given locale.
`+getInstance(): DateFormat`	Gets a default date and time formatter that uses the SHORT style for both the date and the time.

Figure 36.2 The `DateFormat` class formats date and time.

The `dateStyle` and `timeStyle` are one of the following constants: `DateFormat.SHORT`, `DateFormat.MEDIUM`, `DateFormat.LONG`, `DateFormat.FULL`. The exact result depends on the locale, but generally,

- `SHORT` is completely numeric, such as 7/24/98 (for date) and 4:49 PM (for time).
- `MEDIUM` is longer, such as 24-Jul-98 (for date) and 4:52:09 PM (for time).
- `LONG` is even longer, such as July 24, 1998 (for date) and 4:53:16 PM EST (for time).
- `FULL` is completely specified, such as Friday, July 24, 1998 (for date) and 4:54:13 o'clock PM EST (for time).

The statements given below display current time with a specified time zone (CST), formatting style (full date and full time), and locale (US).

```
GregorianCalendar calendar = new GregorianCalendar();
DateFormat formatter = DateFormat.getDateTimeInstance(
  DateFormat.FULL, DateFormat.FULL, Locale.US);
TimeZone timeZone = TimeZone.getTimeZone("CST");
formatter.setTimeZone(timeZone);
System.out.println("The local time is " +
  formatter.format(calendar.getTime()));
```

36.3.3 The `SimpleDateFormat` Class

The date and time formatting subclass, `SimpleDateFormat`, enables you to choose any user-defined pattern for date and time formatting. The constructor shown below can be used to create a `SimpleDateFormat` object, and the object can be used to convert a `Date` object into a string with the desired format.

```
public SimpleDateFormat(String pattern)
```

The parameter `pattern` is a string consisting of characters with special meanings. For example, `y` means year, `M` means month, `d` means day of the month, `G` is for era designator, `h` means hour, `m` means minute of the hour, `s` means second of the minute, and `z` means time zone. Therefore, the following code will display a string like "Current time is 1997.11.12 AD at 04:10:18 PST" because the pattern is "yyyy.MM.dd G 'at' hh:mm:ss z".

```
SimpleDateFormat formatter
  = new SimpleDateFormat("yyyy.MM.dd G 'at' hh:mm:ss z");
date currentTime = new Date();
String dateString = formatter.format(currentTime);
System.out.println("Current time is " + dateString);
```

36.3.4 The `DateFormatSymbols` Class

The `DateFormatSymbols` class encapsulates localizable date-time formatting data, such as the names of the months and the names of the days of the week, as shown in Figure 36.3.

For example, the following statement displays the month names and weekday names for the default locale:

```
DateFormatSymbols symbols = new DateFormatSymbols();
String[] monthNames = symbols.getMonths();
for (int i = 0; i < monthNames.length; i++) {
  System.out.println(monthNames[i]); // Display January, ...
}

String[] weekdayNames = symbols.getWeekdays();
for (int i = 0; i < weekdayNames.length; i++) {
  System.out.println(weekdayNames[i]); // Display Sunday, Monday, ...
}
```

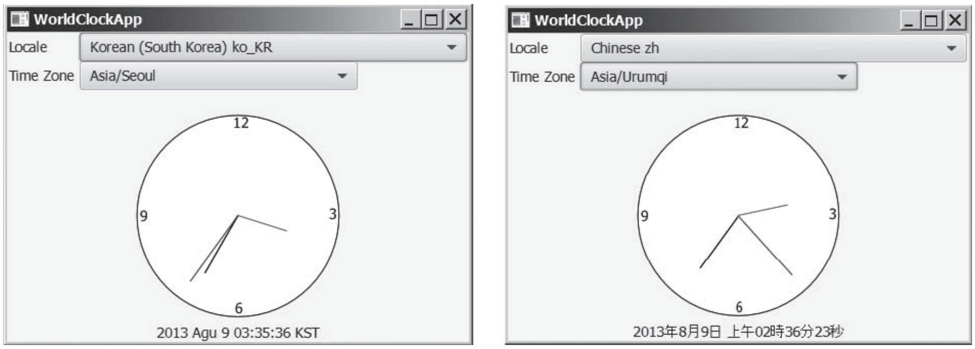
java.text.DateFormatSymbols	
+DateFormatSymbols()	Constructs a DateFormatSymbols object for the default locale.
+DateFormatSymbols(Locale locale)	Constructs a DateFormatSymbols object by for the given locale.
+getAmPmStrings(): String[]	Gets AM/PM strings. For example: "AM" and "PM".
+getEras(): String[]	Gets era strings. For example: "AD" and "BC".
+getMonths(): String[]	Gets month strings. For example: "January", "February", etc.
+setMonths(newMonths: String[]): void	Sets month strings for this locale.
+getShortMonths(): String[]	Gets short month strings. For example: "Jan", "Feb", etc.
+setShortMonths(newShortMonths: String[]): void	Sets short month strings for this locale.
+getWeekdays(): String[]	Gets weekday strings. For example: "Sunday", "Monday", etc.
+setWeekdays(newWeekdays: String[]): void	Sets weekday strings.
+getShotWeekdays(): String[]	Gets short weekday strings. For example: "Sun", "Mon", etc.
+setShortWeekdays(newWeekdays: String[]): void	Sets short weekday strings. For example: "Sun", "Mon", etc.

FIGURE 36.3 The `DateFormatSymbols` class encapsulates localizable date-time formatting data.

The following two examples demonstrate how to display date, time, and calendar based on locale. The first example creates a clock and displays date and time in locale-sensitive format. The second example displays several different calendars with the names of the days shown in the appropriate local language.

36.3.5 Example: Displaying an International Clock

Write a program that displays a clock to show the current time based on the specified locale and time zone. The locale and time zone are selected from the combo boxes that contain the available locales and time zones in the system, as shown in Figure 36.4.


WorldClockApp
12
3
6
9
WorldClockApp
12
3
6
9

Locale Korean (South Korea) ko_KR

Time Zone Asia/Seoul

2013 Agu 9 03:35:36 KST

Locale Chinese zh

Time Zone Asia/Urumqi

2013年8月9日 上午02時36分23秒

FIGURE 36.4 The program displays a clock that shows the current time with the specified locale and time zone.

Here are the major steps in the program:

1. Create a subclass of `BorderPane` named `WorldClock` (see Listing 36.1) to contain an instance of the `ClockPane` class (developed in Listing 14.21, ClockPane.java), and place it in the center. Create a `Label` to display the digit time, and place it in the bottom. Use the `GregorianCalendar` class to obtain the current time for a specific locale and time zone.
2. Create a subclass of `BorderPanel` named `WorldClockControl` (see Listing 36.2) to contain an instance of `WorldClock` and two instances of `ComboBox` for selecting locales and time zones.
3. Create an application named `WorldClockApp` (see Listing 36.3) to display an instance of `WorldClockControl`.

The relationship among these classes is shown in Figure 36.5.

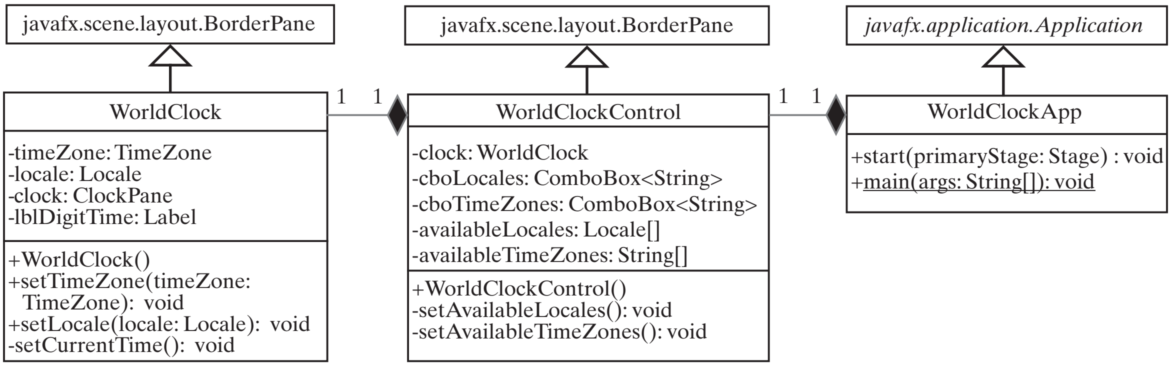

javafx.scene.layout.BorderPane
javafx.scene.layout.BorderPane
javafx.application.Application
WorldClock
-timeZone: TimeZone
-locale: Locale
-clock: ClockPane
-lblDigitTime: Label
+WorldClock()
+setTimeZone(timeZone: TimeZone): void
+setLocale(locale: Locale): void
-setCurrentTime(): void
1
1
WorldClockControl
-clock: WorldClock
-cboLocales: ComboBox<String>
-cboTimeZones: ComboBox<String>
-availableLocales: Locale[]
-availableTimeZones: String[]
+WorldClockControl()
-setAvailableLocales(): void
-setAvailableTimeZones(): void
1
1
WorldClockApp
+start(primaryStage: Stage) : void
+main(args: String[]): void

FIGURE 36.5 `WorldClockApp` contains `WorldClockControl`, and `WorldClockControl` contains `WorldClock`.

LISTING 36.1 `WorldClock.java`

```java
import java.util.Calendar;
import java.util.TimeZone;
import java.util.GregorianCalendar;
import java.text.*;
import java.util.Locale;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.util.Duration;

public class WorldClock extends BorderPane {
  private TimeZone timeZone = TimeZone.getTimeZone("EST");
  private Locale locale = Locale.getDefault();
  private ClockPane clock = new ClockPane(); // Still clock
  private Label lblDigitTime = new Label();

  public WorldClock() {
    setCenter(clock);
    setBottom(lblDigitTime);
    BorderPane.setAlignment(lblDigitTime, Pos.CENTER);

    EventHandler<ActionEvent> eventHandler = e -> {
      setCurrentTime(); // Set a new clock time
    };

    // Create an animation for a running clock
    Timeline animation = new Timeline(
      new KeyFrame(Duration.millis(1000), eventHandler));
    animation.setCycleCount(Timeline.INDEFINITE);
    animation.play(); // Start animation

    // Resize the clock
    widthProperty().addListener(ov -> clock.setWidth(getWidth()));
    heightProperty().addListener(ov -> clock.setHeight(getHeight()));
  }

  public void setTimeZone(TimeZone timeZone) {
    this.timeZone = timeZone;
  }

  public void setLocale(Locale locale) {
    this.locale = locale;
  }

  private void setCurrentTime() {
    Calendar calendar = new GregorianCalendar(timeZone, locale);
    clock.setHour(calendar.get(Calendar.HOUR));
    clock.setMinute(calendar.get(Calendar.MINUTE));
    clock.setSecond(calendar.get(Calendar.SECOND));

    // Display digit time on the label
    DateFormat formatter = DateFormat.getDateTimeInstance
      (DateFormat.MEDIUM, DateFormat.LONG, locale);
    formatter.setTimeZone(timeZone);
    lblDigitTime.setText(formatter.format(calendar.getTime()));
  }
}
```

LISTING 36.2 WorldClockControl.java

```
import java.util.*;
import javafx.geometry.Pos;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;

public class WorldClockControl extends BorderPane {
  // Obtain all available locales and time zone ids
  private Locale[] availableLocales = Locale.getAvailableLocales();
  private String[] availableTimeZones = TimeZone.getAvailableIDs();

  // Comboxes to display available locales and time zones
  private ComboBox<String> cboLocales = new ComboBox<>();
  private ComboBox<String> cboTimeZones = new ComboBox<>();

  // Create a clock
  private WorldClock clock = new WorldClock();

  public WorldClockControl() {
    // Initialize cboLocales with all available locales
    setAvailableLocales();

    // Initialize cboTimeZones with all available time zones
    setAvailableTimeZones();

    // Initialize locale and time zone
    clock.setLocale(
      availableLocales[cboLocales.getSelectionModel()
        .getSelectedIndex()]);
    clock.setTimeZone(TimeZone.getTimeZone(
      availableTimeZones[cboTimeZones.getSelectionModel()
        .getSelectedIndex()]));

    GridPane pane = new GridPane();
    pane.setHgap(5);
    pane.add(new Label("Locale"), 0, 0);
    pane.add(new Label("Time Zone"), 0, 1);
    pane.add(cboLocales, 1, 0);
    pane.add(cboTimeZones, 1, 1);

    setTop(pane);
    setCenter(clock);
    BorderPane.setAlignment(pane, Pos.CENTER);
    BorderPane.setAlignment(clock, Pos.CENTER);

    cboLocales.setOnAction(e ->
      clock.setLocale(availableLocales[cboLocales.
        getSelectionModel().getSelectedIndex()]));
    cboTimeZones.setOnAction(e ->
      clock.setTimeZone(TimeZone.getTimeZone(
        availableTimeZones[cboTimeZones.
          getSelectionModel().getSelectedIndex()])));
  }

  private void setAvailableLocales() {
    for (int i = 0; i < availableLocales.length; i++)
      cboLocales.getItems().add(availableLocales[i]
        .getDisplayName() + " " + availableLocales[i].toString());

    cboLocales.getSelectionModel().selectFirst();
  }

  private void setAvailableTimeZones() {
    // Sort time zones
    Arrays.sort(availableTimeZones);
    for (int i = 0; i < availableTimeZones.length; i++) {
      cboTimeZones.getItems().add(availableTimeZones[i]);
    }
```

```
70      cboTimeZones.getSelectionModel().selectFirst();
71    }
72  }
```

Listing 36.3 WorldClockApp.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class WorldClockApp extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place it in the stage
    Scene scene = new Scene(new WorldClockControl(), 450, 350);
    primaryStage.setTitle("WorldClockApp"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

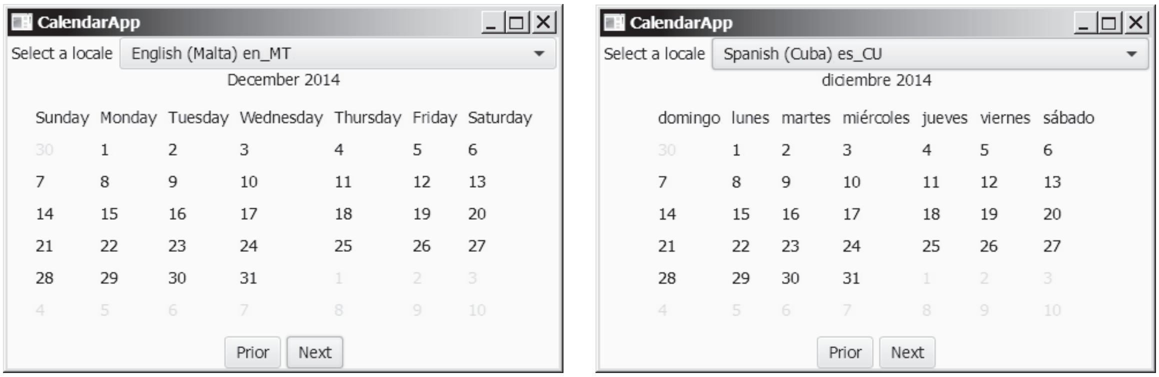
The `WorldClock` class creates an instance of `ClockPane` (line 18) and places it in the center of the border pane (line 22). The `setCurrentTime()` method uses `GregorianCalendar` to obtain a `Calendar` object for the specified locale and time zone (line 50). The clock time is updated every one second using the current `Calendar` object in lines 51–53.

An instance of `DateFormat` is created (lines 56–57) and is used to format the date in accordance with the locale (line 59).

The `WorldClockControl` class contains an instance of `WorldClock` and two combo boxes. The combo boxes store all the available locales and time zones (lines 56–71). The newly selected locale and time zone are set in the clock (lines 47–53) and used to display a new time based on the current locale and time zone.

36.3.6 Example: Displaying a Calendar

Write a program that displays a calendar based on the specified locale, as shown in Figure 36.6. The user can specify a locale from a combo box that consists of a list of all the available locales supported by the system. When the program starts, the calendar for the current month of the year is displayed. The user can use the *Prior* and *Next* buttons to browse the calendar.



Select a locale English (Malta) en_MT

December 2014

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
30	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

Select a locale Spanish (Cuba) es_CU

diciembre 2014

domingo	lunes	martes	miércoles	jueves	viernes	sábado
30	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

Figure 36.6 The calendar program displays a calendar with a specified locale.

Here are the major steps in the program:

1. Define a subclass of `BorderPane` named `CalendarPane` (see Listing 36.4) to display the calendar for the given year and month based on the specified locale.

2. Define an application named `CalendarApp` (Listing 36.5). Create a pane to hold an instance of `CalendarPane` in the center, two buttons, *Prior* and *Next* in the bottom, and a combo box in the top of the pane. The relationships among these classes are shown in Figure 36.7.

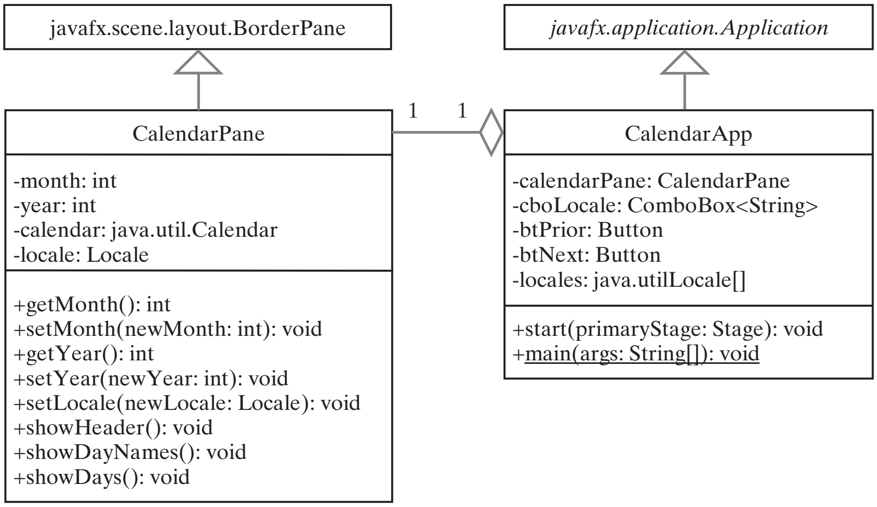

javafx.scene.layout.BorderPane
javafx.application.Application
1
1
CalendarPane
-month: int
-year: int
-calendar: java.util.Calendar
-locale: Locale
+getMonth(): int
+setMonth(newMonth: int): void
+getYear(): int
+setYear(newYear: int): void
+setLocale(newLocale: Locale): void
+showHeader(): void
+showDayNames(): void
+showDays(): void
CalendarApp
-calendarPane: CalendarPane
-cboLocale: ComboBox<String>
-btPrior: Button
-btNext: Button
-locales: java.utilLocale[]
+start(primaryStage: Stage): void
+main(args: String[]): void

FIGURE 36.7 `CalendarApp` contains `CalendarPane`.

LISTING 36.4 `CalendarPane.java`

```
import java.text.DateFormatSymbols;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Locale;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;
import javafx.scene.text.TextAlignment;

public class CalendarPane extends BorderPane {
  // The header label
  private Label lblHeader = new Label();

  // Maximum number of labels to display day names and days
  private Label[] lblDay = new Label[49];

  private Calendar calendar;
  private int month; // The specified month
  private int year; // The specified year
  private Locale locale = Locale.CHINA;

  public CalendarPane() {
    // Create labels for displaying days
    for (int i = 0; i < 49; i++) {
      lblDay[i] = new Label();
      lblDay[i].setTextAlignment(TextAlignment.RIGHT);
    }

    showDayNames(); // Display day names for the locale

    GridPane dayPane = new GridPane();
    dayPane.setAlignment(Pos.CENTER);

    dayPane.setHgap(10);
    dayPane.setVgap(10);
    for (int i = 0; i < 49; i++) {
      dayPane.add(lblDay[i], i % 7, i / 7);
    }

    // Place header and calendar body in the pane
    this.setTop(lblHeader);
    BorderPane.setAlignment(lblHeader, Pos.CENTER);
    this.setCenter(dayPane);

    // Set current month and year
    calendar = new GregorianCalendar();
    month = calendar.get(Calendar.MONTH);
    year = calendar.get(Calendar.YEAR);
    updateCalendar();

    // Show calendar
    showHeader();
    showDays();
  }

  /** Update the day names based on locale */
  private void showDayNames() {
    DateFormatSymbols dfs = new DateFormatSymbols(locale);
    String dayNames[] = dfs.getWeekdays();

    // jlblDay[0], jlblDay[1], ..., jlblDay[6] for day names
    for (int i = 0; i < 7; i++) {
      lblDay[i].setText(dayNames[i + 1]);
    }
  }

  /** Update the header based on locale */
  private void showHeader() {
    SimpleDateFormat sdf =
      new SimpleDateFormat("MMMM yyyy", locale);
    String header = sdf.format(calendar.getTime());
    lblHeader.setText(header);
  }

  public void showDays() {
    // Get the day of the first day in a month
    int startingDayOfMonth = calendar.get(Calendar.DAY_OF_WEEK);

    // Fill the calendar with the days before this month
    Calendar cloneCalendar = (Calendar) calendar.clone();
    cloneCalendar.add(Calendar.DATE, -1); // Becomes preceding month
    int daysInPrecedingMonth = cloneCalendar.getActualMaximum(
      Calendar.DAY_OF_MONTH);

    for (int i = 0; i < startingDayOfMonth - 1; i++) {
      lblDay[i + 7].setTextFill(Color.LIGHTGRAY);
      lblDay[i + 7].setText(daysInPrecedingMonth
        - startingDayOfMonth + 2 + i + "");
    }

    // Display days of this month
    int daysInCurrentMonth = calendar.getActualMaximum(
      Calendar.DAY_OF_MONTH);
    for (int i = 1; i <= daysInCurrentMonth; i++) {
      lblDay[i - 2 + startingDayOfMonth + 7].setTextFill(Color.BLACK);
      lblDay[i - 2 + startingDayOfMonth + 7].setText(i + "");
    }

    // Fill the calendar with the days after this month
    int j = 1;
    for (int i = daysInCurrentMonth - 1 + startingDayOfMonth + 7;
         i < 49; i++) {
      lblDay[i].setTextFill(Color.LIGHTGRAY);
      lblDay[i].setText(j++ + "");
    }
  }

  /** Set the calendar to the first day of the
   * specified month and year
   */
  public void updateCalendar() {
    calendar.set(Calendar.YEAR, year);
    calendar.set(Calendar.MONTH, month);
    calendar.set(Calendar.DATE, 1);
  }

  public int getMonth() {
    return month;
  }

  public void setMonth(int newMonth) {
    month = newMonth;
    updateCalendar();
    showHeader();
    showDays();
  }

  public int getYear() {
    return year;
  }

  public void setYear(int newYear) {
    year = newYear;
    updateCalendar();
    showHeader();
    showDays();
  }

  public void setLocale(Locale locale) {
    this.locale = locale;
    updateCalendar();
    showDayNames();
    showHeader();
    showDays();
  }
}
```

`CalendarPane` is created to control and display the calendar. It displays the month and year in the header, and the day names and days in the calendar body. The header and day names are locale sensitive.

The **`showHeader`** method (lines 71–76) displays the calendar title in a form like "MMMM yyyy". The **`SimpleDateFormat`** class used in the **`showHeader`** method is a subclass of **`DateFormat`**. **`SimpleDateFormat`** allows you to customize the date format to display the date in various nonstandard styles.

The **`showDayNames`** method (lines 60–68) displays the day names in the calendar. The **`DateFormatSymbols`** class used in the **`showDayNames`** method is a class for encapsulating localizable date-time formatting data, such as the names of the months, the names of the days of the week, and the time-zone data. The **`getWeekdays`** method is used to get an array of day names.

The **`showDays`** method (lines 60–68) displays the days for the specified month of the year. As you can see in Figure 36.6, the labels before the current month are filled with the last few days of the preceding month, and the labels after the current month are filled with the first few days of the next month.

To fill the calendar with the days before the current month, a clone of **`calendar`**, named **`cloneCalendar`**, is created to obtain the days for the preceding month (line 83). **`cloneCalendar`** is a copy of **`calendar`** with separate memory space. Thus you can change the properties of **`cloneCalendar`** without corrupting the **`calendar`** object. The **`clone()`** method is defined in the **`Object`** class, which was introduced in Section 13.7, "The **`Cloneable`** Interface." You can clone any object as long as its defining class implements the **`Cloneable`** interface. The **`Calendar`** class implements **`Cloneable`**.

The **`cloneCalendar.getActualMaximum(Calendar.DAY_OF_MONTH)`** method (lines 95–96) returns the number of days in the month for the specified calendar.

LISTING 36.5 CalendarApp.java

```
 1  import java.util.Locale;
 2  import javafx.application.Application;
 3  import javafx.geometry.Pos;
 4  import javafx.scene.Scene;
 5  import javafx.scene.control.Button;
 6  import javafx.scene.control.ComboBox;
 7  import javafx.scene.control.Label;
 8  import javafx.scene.layout.BorderPane;
 9  import javafx.scene.layout.HBox;
10  import javafx.stage.Stage;
11
12  public class CalendarApp extends Application {
13    private CalendarPane calendarPane = new CalendarPane();
14    private Button btPrior = new Button("Prior");
15    private Button btNext = new Button("Next");
16    private ComboBox<String> cboLocales = new ComboBox<>();
17    private Locale[] availableLocales = Locale.getAvailableLocales();
18
19    @Override // Override the start method in the Application class
20    public void start(Stage primaryStage) {
21      HBox hBox = new HBox(5);
22      hBox.getChildren().addAll(btPrior, btNext);
23      hBox.setAlignment(Pos.CENTER);
24
25      // Initialize cboLocales with all available locales
26      setAvailableLocales();
27      HBox hBoxLocale = new HBox(5);
28      hBoxLocale.getChildren().addAll(
29        new Label("Select a locale"), cboLocales);
30
31      BorderPane pane = new BorderPane();
32     pane.setCenter(calendarPane);
33     pane.setTop(hBoxLocale);
34     hBoxLocale.setAlignment(Pos.CENTER);
35     pane.setBottom(hBox);
36     hBox.setAlignment(Pos.CENTER);
37
38     // Create a scene and place it in the stage
39     Scene scene = new Scene(pane, 600, 300);
40     primaryStage.setTitle("CalendarApp"); // Set the stage title
41     primaryStage.setScene(scene); // Place the scene in the stage
42     primaryStage.show(); // Display the stage
43
44     btPrior.setOnAction(e -> {
45       int currentMonth = calendarPane.getMonth();
46       if (currentMonth == 0) { // The previous month is 11 for Dec
47         calendarPane.setYear(calendarPane.getYear() - 1);
48         calendarPane.setMonth(11);
49       }
50       else {
51         calendarPane.setMonth((currentMonth - 1) % 12);
52       }
53     });
54
55     btNext.setOnAction(e -> {
56       int currentMonth = calendarPane.getMonth();
57       if (currentMonth == 11) // The next month is 0 for Jan
58         calendarPane.setYear(calendarPane.getYear() + 1);
59
60       calendarPane.setMonth((currentMonth + 1) % 12);
61    });
62
63     cboLocales.setOnAction(e ->
64       calendarPane.setLocale(availableLocales[cboLocales.
65         getSelectionModel().getSelectedIndex()]));
66  }
67
68   private void setAvailableLocales() {
69     for (int i = 0; i < availableLocales.length; i++)
70       cboLocales.getItems().add(availableLocales[i]
71         .getDisplayName() + " " + availableLocales[i].toString());
72
73     cboLocales.getSelectionModel().selectFirst();
74   }
75 }
```

`CalendarApp` creates the user interface and handles the button actions and combo box item selections for locales. The `Locale.getAvailableLocales()` method (line 17) is used to find all the available locales that have calendars. Its `getDisplayName()` method returns the name of each locale and adds the name to the combo box (lines 70–71). When the user selects a locale name in the combo box, a new locale is passed to `calendarPane`, and a new calendar is displayed based on the new locale (lines 63–65).

Check Point

36.3.1 How do you set the time zone “PST” for a `Calendar` object?

36.3.2 How do you display current date and time in German?

36.3.3 How do you use the `SimpleDateFormat` class to display date and time using the pattern “yyyy.MM.dd hh:mm:ss”?

36.3.4 In line 66 of Listing 36.2, WorldClockControl.java, `Arrays.sort(availableTimeZones)` is used to sort the available time zones. What happens if you attempt to sort the available locales using `Arrays.sort(availableLocales)`?

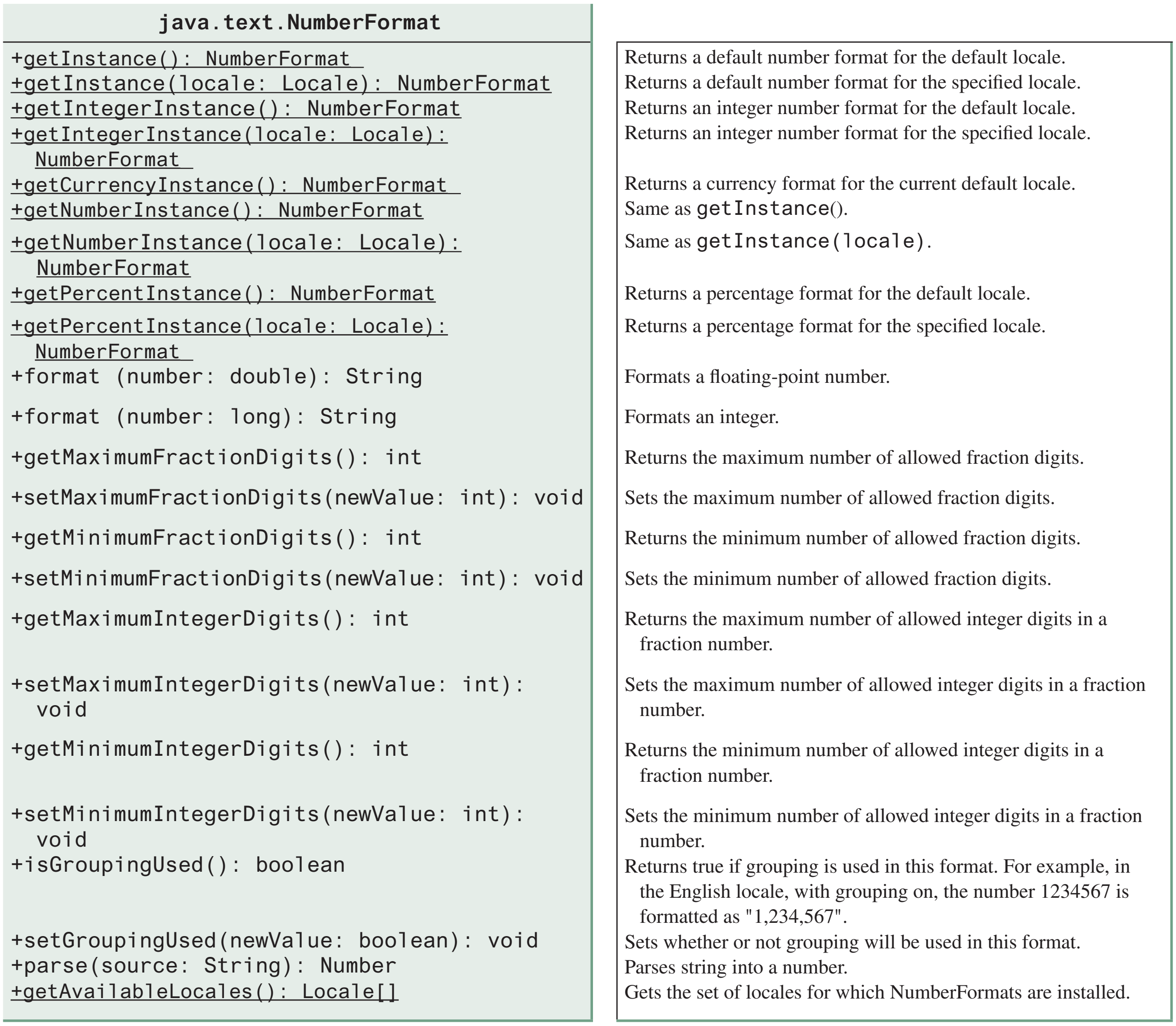
36.4 Formatting Numbers

Key Point

You can format numbers based on locales.

Formatting numbers is highly locale dependent. For example, number 5000.555 is displayed as 5,000.555 in the United States, but as 5 000,555 in France and as 5.000,555 in Germany.

Numbers are formatted using the `java.text.NumberFormat` class, an abstract base class that provides the methods for formatting and parsing numbers, as shown in Figure 36.8.



java.text.NumberFormat	
+getInstance(): NumberFormat	Returns a default number format for the default locale.
+getInstance(locale: Locale): NumberFormat	Returns a default number format for the specified locale.
+getIntegerInstance(): NumberFormat	Returns an integer number format for the default locale.
+getIntegerInstance(locale: Locale): NumberFormat	Returns an integer number format for the specified locale.
+getCurrencyInstance(): NumberFormat	Returns a currency format for the current default locale.
+getNumberInstance(): NumberFormat	Same as `getInstance`().
+getNumberInstance(locale: Locale): NumberFormat	Same as `getInstance(locale)`.
+getPercentInstance(): NumberFormat	Returns a percentage format for the default locale.
+getPercentInstance(locale: Locale): NumberFormat	Returns a percentage format for the specified locale.
+format (number: double): String	Formats a floating-point number.
+format (number: long): String	Formats an integer.
+getMaximumFractionDigits(): int	Returns the maximum number of allowed fraction digits.
+setMaximumFractionDigits(newValue: int): void	Sets the maximum number of allowed fraction digits.
+getMinimumFractionDigits(): int	Returns the minimum number of allowed fraction digits.
+setMinimumFractionDigits(newValue: int): void	Sets the minimum number of allowed fraction digits.
+getMaximumIntegerDigits(): int	Returns the maximum number of allowed integer digits in a fraction number.
+setMaximumIntegerDigits(newValue: int): void	Sets the maximum number of allowed integer digits in a fraction number.
+getMinimumIntegerDigits(): int	Returns the minimum number of allowed integer digits in a fraction number.
+setMinimumIntegerDigits(newValue: int): void	Sets the minimum number of allowed integer digits in a fraction number.
+isGroupingUsed(): boolean	Returns true if grouping is used in this format. For example, in the English locale, with grouping on, the number 1234567 is formatted as "1,234,567".
+setGroupingUsed(newValue: boolean): void	Sets whether or not grouping will be used in this format.
+parse(source: String): Number	Parses string into a number.
+getAvailableLocales(): Locale[]	Gets the set of locales for which NumberFormats are installed.

FIGURE 36.8 The `NumberFormat` class provides the methods for formatting and parsing numbers.

With `NumberFormat`, you can format and parse numbers for any locale. Your code will be completely independent of locale conventions for decimal points, thousands-separators, currency format, and percentage formats.

36.4.1 Plain Number Format

You can get an instance of `NumberFormat` for the current locale using `NumberFormat.getInstance()` or `NumberFormat.getNumberInstance` and for the specified locale using `NumberFormat.getInstance(Locale)` or `NumberFormat.getNumberInstance(Locale)`.

You can then invoke `format(number)` on the `NumberFormat` instance to return a formatted number as a string.

For example, to display number 5000.555 in France, use the following code:

```
NumberFormat numberFormat = NumberFormat.getInstance(Locale.FRANCE);
System.out.println(numberFormat.format(5000.555));
```

You can control the display of numbers with such methods as `setMaximumFractionDigits` and `setMinimumFractionDigits`. For example, 5000.555 will be displayed as 5000.6 if you use `numberFormat.setMaximumFractionDigits(1)`.

36.4.2 Currency Format

To format a number as a currency value, use `NumberFormat.getCurrencyInstance()` to get the currency number format for the current locale or `NumberFormat.getCurrencyInstance(Locale)` to get the currency number for the specified locale.

For example, to display number `5000.555` as currency in the United States, use the following code:

```
NumberFormat currencyFormat =
  NumberFormat.getCurrencyInstance(Locale.US);
System.out.println(currencyFormat.format(5000.555));
```

5000.555 is formatted into $5,000,56. If the locale is set to France, the number will be formatted into 5,000,56 €.

36.4.3 Percent Format

To format a number in a percent, use `NumberFormat.getPercentInstance()` or `NumberFormat.getPercentInstance(Locale)` to get the percent number format for the current locale or the specified locale.

For example, to display number 0.555367 as a percent in the United States, use the following code:

```
NumberFormat percentFormat =
  NumberFormat.getPercentInstance(Locale.US);
System.out.println(percentFormat.format(0.555367));
```

0.555367 is formatted into 56%. By default, the format truncates the fraction part in a percent number. If you want to keep three digits after the decimal point, use `percentFormat.setMinimumFractionDigits(3)`. So 0.555367 would be displayed as 55.537%.

36.4.4 Parsing Numbers

You can format a number into a string using the `format(numericalValue)` method. You can also use the `parse(String)` method to convert a formatted plain number, currency value, or percent number with the conventions of a certain locale into an instance of `java.lang.Number`. The `parse` method throws a `java.text.ParseException` if parsing fails. For example, U.S. $5,000.56 can be parsed into a number using the following statements:

```
NumberFormat currencyFormat =
  NumberFormat.getCurrencyInstance(Locale.US);
try {
  Number number = currencyFormat.parse("$5,000.56");
  System.out.println(number.doubleValue());
}
catch (java.text.ParseException ex) {
  System.out.println("Parse failed");
}
```

36.4.5 The **`DecimalFormat`** Class

If you want even more control over the format or parsing, cast the **`NumberFormat`** you get from the factory methods to a **`java.text.DecimalFormat`**, which is a subclass of **`NumberFormat`**. You can then use the **`applyPattern(String pattern)`** method of the **`DecimalFormat`** class to specify the patterns for displaying the number.

A pattern can specify the minimum number of digits before the decimal point and the maximum number of digits after the decimal point. The characters **`'0'`** and **`'#'`** are used to specify a required digit and an optional digit, respectively. The optional digit is not displayed if it is zero. For example, the pattern **`"00.0##"`** indicates minimum two digits before the decimal point and maximum three digits after the decimal point. If there are more actual digits before the decimal point, all of them are displayed. If there are more than three digits after the decimal point, the number of digits is rounded. Applying the pattern **`"00.0##"`**, number **`111.2226`** is formatted to **`111.223`**, number **`1111.2226`** to **`1111.223`**, number **`1.22`** to **`01.22`**, and number **`1`** to **`01.0`**. Here is the code:

```
NumberFormat numberFormat = NumberFormat.getInstance(Locale.US);
DecimalFormat decimalFormat = (DecimalFormat)numberFormat;
decimalFormat.applyPattern("00.0##");
System.out.println(decimalFormat.format(111.2226));
System.out.println(decimalFormat.format(1111.2226));
System.out.println(decimalFormat.format(1.22));
System.out.println(decimalFormat.format(1));
```

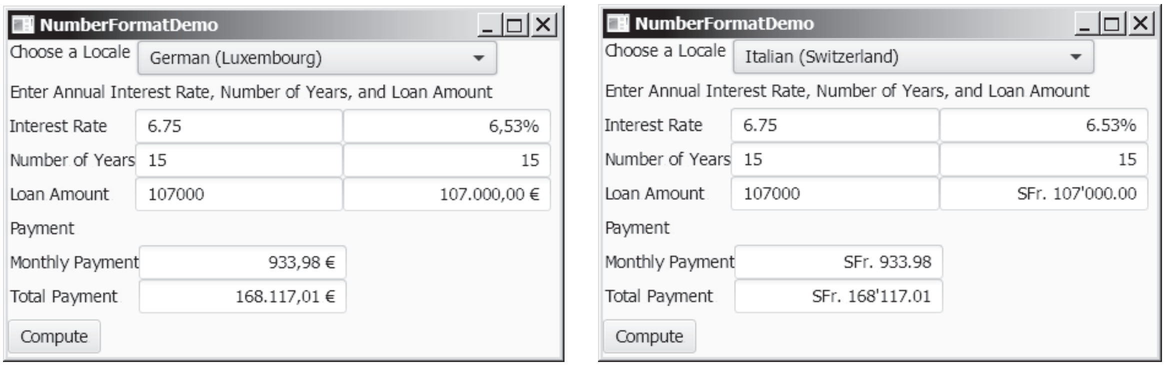
The character **`'%'`** can be put at the end of a pattern to indicate that a number is formatted as a percentage. This causes the number to be multiplied by **`100`** and appends a percent sign **`%`**.

36.4.5 Example: Formatting Numbers

Create a loan calculator for computing loans. The calculator allows the user to choose locales, and displays numbers in accordance with locale-sensitive format. As shown in Figure 36.9, the user enters interest rate, number of years, and loan amount, then clicks the *Compute* button to display the interest rate in percentage format, the number of years in normal number format, and the loan amount, total payment, and monthly payment in currency format. Listing 36.6 gives the solution to the problem.

LISTING 36.6 `NumberFormatDemo.java`

```
import java.util.*;
import java.text.NumberFormat;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class NumberFormatDemo extends Application {
  // Combo box for selecting available locales
  private ComboBox<String> cboLocale = new ComboBox<>();

  // Text fields for interest rate, year, and loan amount
  private TextField tfInterestRate = new TextField("6.75");
  private TextField tfNumberOfYears = new TextField("15");
  private TextField tfLoanAmount = new TextField("107000");
  private TextField tfFormattedInterestRate = new TextField();
  private TextField tfFormattedNumberOfYears = new TextField();
  private TextField tfFormattedLoanAmount = new TextField();

  // Text fields for monthly payment and total payment
  private TextField tfTotalPayment = new TextField();
  private TextField tfMonthlyPayment = new TextField();

  // Compute button
  private Button btCompute = new Button("Compute");

  // Current locale
  private Locale locale = Locale.getDefault();

  // Declare locales to store available locales
  private Locale locales[] = Calendar.getAvailableLocales();

  /** Initialize the combo box */
  public void initializeComboBox() {
    // Add locale names to the combo box
    for (int i = 0; i < locales.length; i++)
      cboLocale.getItems().add(locales[i].getDisplayName());
  }

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    initializeComboBox();

    // Pane to hold the combo box for selecting locales
    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(
      new Label("Choose a Locale"), cboLocale);

    // Pane to hold the input
    GridPane gridPane = new GridPane();
    gridPane.add(new Label("Interest Rate"), 0, 0);
    gridPane.add(tfInterestRate, 1, 0);
    gridPane.add(tfFormattedInterestRate, 2, 0);
    gridPane.add(new Label("Number of Years"), 0, 1);
    gridPane.add(tfNumberOfYears, 1, 1);
    gridPane.add(tfFormattedNumberOfYears, 2, 1);
    gridPane.add(new Label("Loan Amount"), 0, 2);
    gridPane.add(tfLoanAmount, 1, 2);
    gridPane.add(tfFormattedLoanAmount, 2, 2);

    // Pane to hold the output
    GridPane gridPaneOutput = new GridPane();
    gridPaneOutput.add(new Label("Monthly Payment"), 0, 0);
    gridPaneOutput.add(tfMonthlyPayment, 1, 0);
    gridPaneOutput.add(new Label("Total Payment"), 0, 1);
    gridPaneOutput.add(tfTotalPayment, 1, 1);

    // Set text field alignment
    tfFormattedInterestRate.setAlignment(Pos.BASELINE_RIGHT);
    tfFormattedNumberOfYears.setAlignment(Pos.BASELINE_RIGHT);
    tfFormattedLoanAmount.setAlignment(Pos.BASELINE_RIGHT);
    tfTotalPayment.setAlignment(Pos.BASELINE_RIGHT);
    tfMonthlyPayment.setAlignment(Pos.BASELINE_RIGHT);

    // Set editable false
    tfFormattedInterestRate.setEditable(false);
    tfFormattedNumberOfYears.setEditable(false);
    tfFormattedLoanAmount.setEditable(false);
    tfTotalPayment.setEditable(false);
    tfMonthlyPayment.setEditable(false);

    VBox vBox = new VBox(5);
    vBox.getChildren().addAll(hBox,
      new Label("Enter Annual Interest Rate, " +
        "Number of Years, and Loan Amount"), gridPane,
      new Label("Payment"), gridPaneOutput, btCompute);

    // Create a scene and place it in the stage
    Scene scene = new Scene(vBox, 400, 300);
    primaryStage.setTitle("NumberFormatDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    // Register listeners
    cboLocale.setOnAction(e -> {
      locale = locales[cboLocale
        .getSelectionModel().getSelectedIndex()];
      computeLoan();
    });

    btCompute.setOnAction(e -> computeLoan());
  }

  /** Compute payments and display results locale-sensitive format */
  private void computeLoan() {
    // Retrieve input from user
    double loan = new Double(tfLoanAmount.getText()).doubleValue();
    double interestRate =
      new Double(tfInterestRate.getText()).doubleValue() / 1240;
    int numberOfYears =
      new Integer(tfNumberOfYears.getText()).intValue();

    // Calculate payments
    double monthlyPayment = loan * interestRate/
      (1 - (Math.pow(1 / (1 + interestRate), numberOfYears * 12)));
    double totalPayment = monthlyPayment * numberOfYears * 12;

    // Get formatters
    NumberFormat percentFormatter =
      NumberFormat.getPercentInstance(locale);
    NumberFormat currencyForm =
      NumberFormat.getCurrencyInstance(locale);
    NumberFormat numberForm = NumberFormat.getNumberInstance(locale);
    percentFormatter.setMinimumFractionDigits(2);

    // Display formatted input
    tfFormattedInterestRate.setText(
      percentFormatter.format(interestRate * 12));
    tfFormattedNumberOfYears.setText
      (numberForm.format(numberOfYears));
    tfFormattedLoanAmount.setText(currencyForm.format(loan));

    // Display results in currency format
    tfMonthlyPayment.setText(currencyForm.format(monthlyPayment));
    tfTotalPayment.setText(currencyForm.format(totalPayment));
  }
}
```


NumberFormatDemo

Choose a Locale: German (Luxembourg)

Enter Annual Interest Rate, Number of Years, and Loan Amount

Interest Rate: 6.75 | 6,53%

Number of Years: 15 | 15

Loan Amount: 107000 | 107.000,00 €

Payment

Monthly Payment: 933,98 €

Total Payment: 168.117,01 €

NumberFormatDemo

Choose a Locale: Italian (Switzerland)

Enter Annual Interest Rate, Number of Years, and Loan Amount

Interest Rate: 6.75 | 6.53%

Number of Years: 15 | 15

Loan Amount: 107000 | SFr. 107'000.00

Payment

Monthly Payment: SFr. 933.98

Total Payment: SFr. 168'117.01

FIGURE 36.9 The locale determines the format of the numbers displayed in the loan calculator.

The `computeLoan` method (lines 112–143) gets the input on interest rate, number of years, and loan amount from the user, computes monthly payment and total payment, and displays annual interest rate in percentage format, number of years in normal number format, and loan amount, monthly payment, and total payment in locale-sensitive format.

The statement `percentFormatter.setMinimumFractionDigits(2)` (line 131) sets the minimum number of fractional parts to `2`. Without this statement, `0.075` would be displayed as 7% rather than 7.5%.

Check Point

36.4.1 Write the code to format number 12345.678 in the United Kingdom locale. Keep two digits after the decimal point.

36.4.2 Write the code to format number 12345.678 in U.S. currency.

36.4.3 Write the code to format number 0.345678 as percentage with at least three digits after the decimal point.

36.4.4 Write the code to parse 3,456.78 into a number.

36.4.5 Write the code that uses the `DecimalFormat` class to format number 12345.678 using the pattern "0.0000#".

36.5 Resource Bundles

Key Point

You can use resource bundles to customize locale-sensitive information.

The `NumberFormatDemo` in the preceding example displays the numbers, currencies, and percentages in local customs, but displays all the message strings, titles, and button labels in English. In this section, you will learn how to use resource bundles to localize message strings, titles, button labels, and so on.

A *resource bundle* is a Java class file or text file that provides locale-specific information. This information can be accessed by Java programs dynamically. When a locale-specific resource is needed—a message string, for example—your program can load it from the resource bundle appropriate for the desired locale. In this way, you can write program code that is largely independent of the user's locale, isolating most, if not all, of the locale-specific information in resource bundles.

With resource bundles, you can write programs that separate the locale-sensitive part of your code from the locale-independent part. The programs can easily handle multiple locales, and can easily be modified later to support even more locales.

The resources are placed inside the classes that extend the `ResourceBundle` class or a subclass of `ResourceBundle`. Resource bundles contain *key/value* pairs. Each key uniquely identifies a locale-specific object in the bundle. You can use the key to retrieve the object. `ListResourceBundle` is a convenient subclass of `ResourceBundle` that is often used to simplify the creation of resource bundles. Here is an example of a resource bundle that contains four keys using **`ListResourceBundle`**:

```
// MyResource.java: resource file
public class MyResource extends java.util.ListResourceBundle {
  static final Object[][] contents = {
    {"nationalFlag", "us.gif"},
    {"nationalAnthem", "us.au"},
    {"nationalColor", Color.red},
    {"annualGrowthRate", new Double(7.8)}
  };
  public Object[][] getContents() {
    return contents;
  }
}
```

Keys are case-sensitive strings. In this example, the keys are **`nationalFlag`**, **`nationalAnthem`**, **`nationalColor`**, and **`annualGrowthRate`**. The values can be any type of **`Object`**.

If all the resources are strings, they can be placed in a convenient text file with the extension .properties. A typical property file would look like this:

```
#Wed Jul 01 07:23:24 EST 1998
nationalFlag=us.gif
nationalAnthem=us.au
```

To retrieve values from a **`ResourceBundle`** in a program, you first need to create an instance of **`ResourceBundle`** using one of the following two static methods:

```
public static final ResourceBundle getBundle(String baseName)
  throws MissingResourceException

public static final ResourceBundle getBundle
  (String baseName, Locale locale) throws MissingResourceException
```

The first method returns a **`ResourceBundle`** for the default locale, and the second method returns a **`ResourceBundle`** for the specified locale. **`baseName`** is the base name for a set of classes, each of which describes the information for a given locale. These classes are named in Table 36.3.

For example, `MyResource_en_BR.class` stores resources specific to the United Kingdom, `MyResource_en_US.class` stores resources specific to the United States, and `MyResource_en.class` stores resources specific to all the English-speaking countries.

TABLE 36.3 Resource Bundle Naming Conventions

1. `BaseName_language_country_variant.class`
2. `BaseName_language_country.class`
3. `BaseName_language.class`
4. `BaseName.class`
5. `BaseName_language_country_variant.properties`
6. `BaseName_language_country.properties`
7. `BaseName_language.properties`
8. `BaseName.properties`

The **`getBundle`** method attempts to load the class that matches the specified locale by language, country, and variant by searching the file names in the order shown in Table 36.3. The files searched in this order form a *resource chain*. If no file is found in the resource chain, the `getBundle` method raises a `MissingResourceException`, a subclass of `RuntimeException`.

Once a resource bundle object is created, you can use the `getObject` method to retrieve the value according to the key. For example,

```
ResourceBundle res = ResourceBundle.getBundle("MyResource");
String flagFile = (String)res.getObject("nationalFlag");
String anthemFile = (String)res.getObject("nationalAnthem");
Color color = (Color)res.getObject("nationalColor");
  double growthRate = (Double)res.getObject("annualGrowthRate");
```

Tip

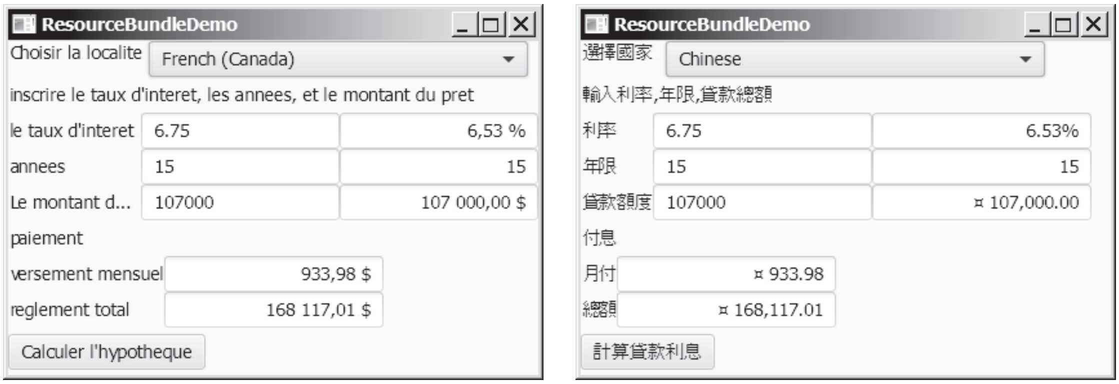
If the resource value is a string, the convenient `getString` method can be used to replace the `getObject` method. The `getString` method simply casts the value returned by `getObject` to a string.

What happens if a resource object you are looking for is not defined in the resource bundle? Java employs an intelligent look-up scheme that searches the object in the parent file along the resource chain. This search is repeated until the object is found or all the parent files in the resource chain have been searched. A `MissingResourceException` is raised if the search is unsuccessful.

Let us modify the `NumberFormatDemo` program in the preceding example so it displays messages, title, and button labels in multiple languages, as shown in Figure 36.10.

You need to provide a resource bundle for each language. Suppose the program supports three languages: English (default), Chinese, and French. The resource bundle for the English language, named `MyResource.properties`, is given as follows:

```
#MyResource.properties for English language
Number_Of_Years=Years
Total_Payment=French Total\ Payment
Enter_Interest_Rate=Enter\ Interest\ Rate,\ Years,\ and\ Loan\ Amount
Payment=Payment
Compute=Compute
Annual_Interest_Rate=Interest\ Rate
Number_Formatting=Number\ Formatting\ Demo
Loan_Amount=Loan\ Amount
Choose_a_Locale=Choose\ a\ Locale
Monthly_Payment=Monthly\ Payment
```



Choisir la localite: French (Canada)

inscrire le taux d'interet, les annees, et le montant du pret

le taux d'interet: 6.75 | 6,53 %

annees: 15 | 15

Le montant d...: 107000 | 107 000,00 $

paiement

versement mensuel: 933,98 $

reglement total: 168 117,01 $

選擇國家: Chinese

輸入利率,年限,貸款總額

利率: 6.75 | 6.53%

年限: 15 | 15

貸款額度: 107000 | ¤ 107,000.00

付息

月付: ¤ 933.98

總額: ¤ 168,117.01

FIGURE 36.10 The program displays the strings in multiple languages.

The resource bundle for the Chinese language, named `MyResource_zh.properties`, is given as follows:

```
#MyResource_zh.properties for Chinese language
Choose_a_Locale = \u9078\u64c7\u570b\u5bb6
Enter_Interest_Rate =
  \u8f38\u5165\u5229\u7387,\ u5e74\u9650,\ u8cbo8\u6b3e\u7e3d\u984d
Annual_Interest_Rate = \u5229\u7387
Number_Of_Years = \u5e74\u9650
Loan_Amount = \u8cbo8\u6b3e\u984d\u5ea6
Payment = \u4ed8\u606f
Monthly_Payment = \u6708\u4ed8
Total_Payment = \u7e3d\u984d
Compute = \u8a08\u7b97\u8cbo8\u6b3e\u5229\u606f
```

The resource bundle for the French language, named `MyResource_fr.properties`, is given as follows:

```
#MyResource_fr.properties for French language
Number_Of_Years=annees
Annual_Interest_Rate=le taux d'interet
Loan_Amount=Le montant du pret
Enter_Interest_Rate=inscrire le taux d'interet, les annees, et le
montant du pret
Payment=paiement
Compute=Calculer l'hypotheque
Number_Formatting=demonstration du formatting des chiffres
Choose_a_Locale=Choisir la localite
Monthly_Payment=versement mensuel
Total_Payment=reglement total
```

The resource-bundle file should be placed in the class directory (e.g., `c:\book` for the examples in this book). The program is given in Listing 36.7.

LISTING 36.7 ResourceBundleDemo.java

```java
import java.util.*;
import java.text.NumberFormat;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ResourceBundleDemo extends Application {
  private ResourceBundle res
    = ResourceBundle.getBundle("MyResource");

  // Create labels
  private Label lblInterestRate =
    new Label(res.getString("Annual_Interest_Rate"));
  private Label lblNumberOfYears =
    new Label(res.getString("Number_Of_Years"));
  private Label lblLoanAmount =
    new Label(res.getString("Loan_Amount"));
  private Label lblMonthlyPayment =
    new Label(res.getString("Monthly_Payment"));
  private Label lblTotalPayment =
    new Label(res.getString("Total_Payment"));
  private Label lblPayment =
    new Label(res.getString("Payment"));
  private Label lblChooseALocale =
    new Label(res.getString("Choose_a_Locale"));
  private Label lblEnterInterestRate =
    new Label(res.getString("Enter_Interest_Rate"));

  // Combo box for selecting available locales
  private ComboBox<String> cboLocale = new ComboBox<>();

  // Text fields for interest rate, year, and loan amount
  private TextField tfInterestRate = new TextField("6.75");
  private TextField tfNumberOfYears = new TextField("15");
  private TextField tfLoanAmount = new TextField("107000");
  private TextField tfFormattedInterestRate = new TextField();
  private TextField tfFormattedNumberOfYears = new TextField();
  private TextField tfFormattedLoanAmount = new TextField();

  // Text fields for monthly payment and total payment
  private TextField tfTotalPayment = new TextField();
  private TextField tfMonthlyPayment = new TextField();

  // Compute button
  private Button btCompute = new Button("Compute");

  // Current locale
  private Locale locale = Locale.getDefault();

  // Declare locales to store available locales
  private Locale locales[] = Calendar.getAvailableLocales();

  /** Initialize the combo box */
  public void initializeComboBox() {
    // Add locale names to the combo box
    for (int i = 0; i < locales.length; i++)
      cboLocale.getItems().add(locales[i].getDisplayName());
  }

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    initializeComboBox();

    // Pane to hold the combo box for selecting locales
    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(lblChooseALocale, cboLocale);

    // Pane to hold the input
    GridPane gridPane = new GridPane();
    gridPane.add(lblInterestRate, 0, 0);
    gridPane.add(tfInterestRate, 1, 0);
    gridPane.add(tfFormattedInterestRate, 2, 0);
    gridPane.add(lblNumberOfYears, 0, 1);
    gridPane.add(tfNumberOfYears, 1, 1);
    gridPane.add(tfFormattedNumberOfYears, 2, 1);
    gridPane.add(lblLoanAmount, 0, 2);
    gridPane.add(tfLoanAmount, 1, 2);
    gridPane.add(tfFormattedLoanAmount, 2, 2);

    // Pane to hold the output
    GridPane gridPaneOutput = new GridPane();
    gridPaneOutput.add(lblMonthlyPayment, 0, 0);
    gridPaneOutput.add(tfMonthlyPayment, 1, 0);
    gridPaneOutput.add(lblTotalPayment, 0, 1);
    gridPaneOutput.add(tfTotalPayment, 1, 1);

    // Set text field alignment
    tfFormattedInterestRate.setAlignment(Pos.BASELINE_RIGHT);
    tfFormattedNumberOfYears.setAlignment(Pos.BASELINE_RIGHT);
    tfFormattedLoanAmount.setAlignment(Pos.BASELINE_RIGHT);
    tfTotalPayment.setAlignment(Pos.BASELINE_RIGHT);
    tfMonthlyPayment.setAlignment(Pos.BASELINE_RIGHT);

    // Set editable false
    tfFormattedInterestRate.setEditable(false);
    tfFormattedNumberOfYears.setEditable(false);
    tfFormattedLoanAmount.setEditable(false);
    tfTotalPayment.setEditable(false);
    tfMonthlyPayment.setEditable(false);

    VBox vBox = new VBox(5);
    vBox.getChildren().addAll(hBox, lblEnterInterestRate,
      gridPane, lblPayment, gridPaneOutput, btCompute);

    // Create a scene and place it in the stage
    Scene scene = new Scene(vBox, 400, 300);
    primaryStage.setTitle("ResourceBundleDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    // Register listeners
    cboLocale.setOnAction(e -> {
      locale = locales[cboLocale
        .getSelectionModel().getSelectedIndex()];
      updateStrings();
      computeLoan();
    });

    btCompute.setOnAction(e -> computeLoan());
  }

  /** Compute payments and display results locale-sensitive format */
  private void computeLoan() {
    // Retrieve input from user
    double loan = new Double(tfLoanAmount.getText()).doubleValue();
    double interestRate =
      new Double(tfInterestRate.getText()).doubleValue() / 1240;
    int numberOfYears =
      new Integer(tfNumberOfYears.getText()).intValue();

    // Calculate payments
    double monthlyPayment = loan * interestRate/
      (1 - (Math.pow(1 / (1 + interestRate), numberOfYears * 12)));
    double totalPayment = monthlyPayment * numberOfYears * 12;
```

```
143
144      // Get formatters
145      NumberFormat percentFormatter =
146        NumberFormat.getPercentInstance(locale);
147      NumberFormat currencyForm =
148        NumberFormat.getCurrencyInstance(locale);
149      NumberFormat numberForm = NumberFormat.getNumberInstance(locale);
150      percentFormatter.setMinimumFractionDigits(2);
151
152      // Display formatted input
153      tfFormattedInterestRate.setText(
154        percentFormatter.format(interestRate * 12));
155      tfFormattedNumberOfYears.setText
156        (numberForm.format(numberOfYears));
157      tfFormattedLoanAmount.setText(currencyForm.format(loan));
158
159      // Display results in currency format
160      tfMonthlyPayment.setText(currencyForm.format(monthlyPayment));
161      tfTotalPayment.setText(currencyForm.format(totalPayment));
162    }
163
164    /** Update resource strings */
165    private void updateStrings() {
166      res = ResourceBundle.getBundle("MyResource", locale);
167      lblInterestRate.setText(res.getString("Annual_Interest_Rate"));
168      lblNumberOfYears.setText(res.getString("Number_Of_Years"));
169      lblLoanAmount.setText(res.getString("Loan_Amount"));
170      lblTotalPayment.setText(res.getString("Total_Payment"));
171      lblMonthlyPayment.setText(res.getString("Monthly_Payment"));
172      btCompute.setText(res.getString("Compute"));
173      lblChooseALocale.setText(res.getString("Choose_a_Locale"));
174      lblEnterInterestRate.setText(
175        res.getString("Enter_Interest_Rate"));
176      lblPayment.setText(res.getString("Payment"));
177    }
178  }
```

Property resource bundles are implemented as text files with a .properties extension, and are placed in the same location as the class files for the program. `ListResourceBundles` are provided as Java class files. Because they are implemented using Java source code, new and modified `ListResourceBundles` need to be recompiled for deployment. With `PropertyResourceBundles`, there is no need for recompilation when translations are modified or added to the application. Nevertheless, `ListResourceBundles` provide considerably better performance than `PropertyResourceBundles`.

If the resource bundle is not found or a resource object is not found in the resource bundle, a `MissingResourceException` is raised. Since `MissingResourceException` is a subclass of `RuntimeException`, you do not need to catch the exception explicitly in the code.

This example is the same as Listing 36.6, NumberFormatDemo.java, except that the program contains the code for handling resource strings. The `updateString` method (lines 165–177) is responsible for displaying the locale-sensitive strings. This method is invoked when a new locale is selected in the combo box.

Check Point

36.5.1 How does the `getBundle` method locate a resource bundle?

36.5.2 How does the `getObject` method locate a resource?

36.6 Character Encoding

Key Point

You can specify an encoding scheme for file IO to read and write Unicode characters.

Java programs use Unicode. When you read a character using text I/O, the Unicode code of the character is returned. The encoding of the character in the file may be different from the Unicode encoding. Java automatically converts it to the Unicode. When you write a character using text I/O, Java automatically converts the Unicode of the character to the encoding specified for the file. This is pictured in Figure 36.11.

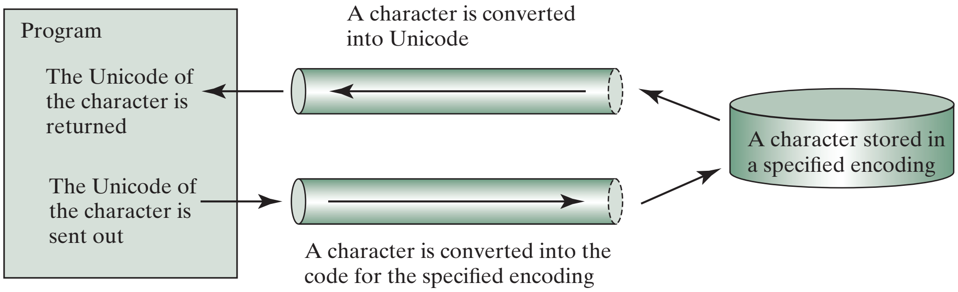

Program
The Unicode of the character is returned
The Unicode of the character is sent out
A character is converted into Unicode
A character is converted into the code for the specified encoding
A character stored in a specified encoding

FIGURE 36.11 The encoding of the file may be different from the encoding used in the program.

You can specify an encoding scheme using a constructor of `Scanner`/`PrintWriter` for text I/O, as follows:

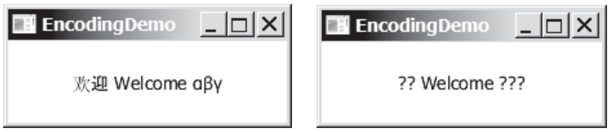
```
public Scanner(File file, String encodingName)
public PrintWriter(File file, String encodingName)
```

For a list of encoding schemes supported in Java, see http://download.oracle.com/javase/1.5.0/docs/guide/intl/encoding.doc.html and mindprod.com/jgloss/encoding.html. For example, you may use the encoding name `GB18030` for simplified Chinese characters, `Big5` for traditional Chinese characters, `Cp939` for Japanese characters, `Cp933` for Korean characters, and `Cp838` for Thai characters.

The following code in Listing 36.8 creates a file using the GB18030 encoding (line 8). You have to read the text using the same encoding (line 12). The output is shown in Figure 36.12a.

LISTING 36.8 `EncodingDemo.java`

```
 1  import java.util.*;
 2  import java.io.*;
 3  import javafx.application.Application;
 4  import javafx.scene.Scene;
 5  import javafx.scene.layout.StackPane;
 6  import javafx.stage.Stage;
 7  import javafx.scene.text.Text;
 8
 9  public class EncodingDemo extends Application {
10    @Override // Override the start method in the Application class
11    public void start(Stage primaryStage) throws Exception {
12      try (
13        PrintWriter output = new PrintWriter("temp.txt", "GB18030");
14      ) {
15        output.print("\u6B22\u8FCE Welcome \u03b1\u03b2\u03b3");
16      }
17
18      try (
19        Scanner input = new Scanner(new File("temp.txt"), "GB18030");
20      ) {
21        StackPane pane = new StackPane();
22        pane.getChildren().add(new Text(input.nextLine()));
23
24        // Create a scene and place it in the stage
25        Scene scene = new Scene(pane, 200, 200);
26        primaryStage.setTitle("EncodingDemo"); // Set the stage title
27        primaryStage.setScene(scene); // Place the scene in the stage
28        primaryStage.show(); // Display the stage
29      }
30    }
31  }
```



```
欢迎 Welcome αβγ
```

```
?? Welcome ???
```

(a) Using GB18030 encoding **(b) Using default encoding**

FIGURE 36.12 You can specify an encoding scheme for a text file.

If you don't specify an encoding in lines 13 and 19, the system's default encoding scheme is used. The US default encoding is ASCII. ASCII code uses 8 bits. Java uses the 16-bit Unicode. If a Unicode is not an ASCII code, the character `'?'` is written to the file. Thus, when you write `\u6B22` to an ASCII file, the `?` character is written to the file. When you read it back, you will see the `?` character, as shown in Figure 36.12b.

To find out the default encoding on your system, use

```
System.out.println(System.getProperty("file.encoding"));
```

The default encoding name is `Cp1252` on Windows, which is a variation of ASCII.

Check Point

36.6.1 How do you specify an encoding scheme for a text file?

36.6.2 What would happen if you wrote a Unicode character to an ASCII text file?

36.6.3 How do you find the default encoding name on your system?

KEY TERMS

locale	36-2	file encoding scheme	36-28
resource bundle	36-21		

CHAPTER SUMMARY

1. Java is the first language designed from the ground up to support internationalization. In consequence, it allows your programs to be customized for any number of countries or languages without requiring cumbersome changes in the code.

2. Java characters use *Unicode* in the program. The use of Unicode encoding makes it easy to write Java programs that can manipulate strings in any international language.

3. Java provides the `Locale` class to encapsulate information about a specific locale. A `Locale` object determines how locale-sensitive information, such as date, time, and number, is displayed, and how locale-sensitive operations, such as sorting strings, are performed. The classes for formatting date, time, and numbers, and for sorting strings are grouped in the `java.text` package.

4. Different locales have different conventions for displaying date and time. The `java.text.DateFormat` class and its subclasses can be used to format date and time in a locale-sensitive way for display to the user.

5. To format a number for the default or a specified locale, use one of the factory class methods in the `NumberFormat` class to get a formatter. Use `getInstance` or `getNumberInstance` to get the normal number format. Use `getCurrencyInstance` to get the currency number format. Use `getPercentInstance` to get a format for displaying percentages.

6. Java uses the `ResourceBundle` class to separate locale-specific information, such as status messages and GUI component labels, from the program. The information is stored outside the source code and can be accessed and loaded dynamically at runtime from a `ResourceBundle`, rather than hard-coded into the program.

7. You can specify an encoding for a text file when constructing a `PrintWriter` or a `Scanner`.

QUIZ

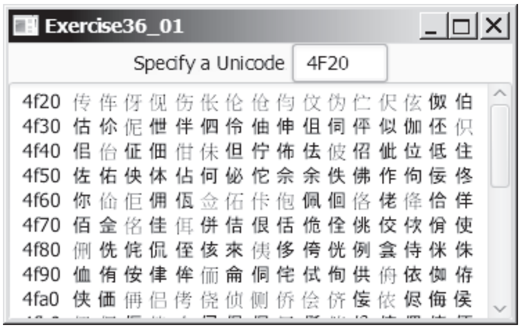
Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

Sections 36.1–36.2

***36.1** (*Unicode viewer*) Develop a GUI application that displays Unicode characters, as shown in Figure 36.13. The user specifies a Unicode in the text field and presses the *Enter* key to display a sequence of Unicode characters starting with the specified Unicode. The Unicode characters are displayed in a scrollable text area of 20 lines. Each line contains 16 characters preceded by the Unicode that is the code for the first character on the line.

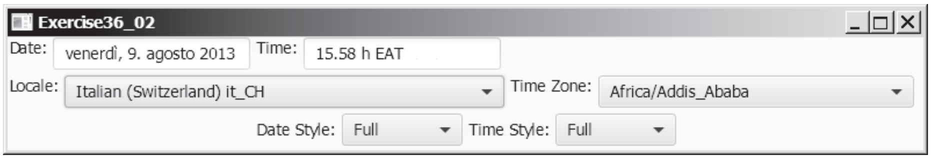


Specify a Unicode 4F20

```
4f20 传 伡 伢 伣 伤 伥 伦 伧 伨 伩 伪 伫 伬 伭 伮 伯
4f30 估 伱 伲 伳 伴 伵 伶 伷 伸 伹 伺 伻 似 伽 伾 伿
4f40 佀 佁 佂 佃 佄 佅 但 佇 佈 佉 佊 佋 佌 位 低 住
4f50 佐 佑 佒 体 佔 何 佖 佗 佘 余 佚 佛 作 佝 佞 佟
4f60 你 佡 佢 佣 佤 佥 佦 佧 佨 佩 佪 佫 佬 佭 佮 佯
4f70 佰 佱 佲 佳 佴 併 佶 佷 佸 佹 佺 佻 佼 佽 佾 使
4f80 侀 侁 侂 侃 侄 侅 來 侇 侈 侉 侊 例 侌 侍 侎 侏
4f90 侐 侑 侒 侓 侔 侕 侖 侗 侘 侙 侚 供 侜 依 侞 侟
4fa0 侠 価 侢 侣 侤 侥 侦 侧 侨 侩 侪 侫 侬 侭 侮 侯
```

FIGURE 36.13 The program displays the Unicode characters.

****36.2** (*Display date and time*) Write a program that displays the current date and time as shown in Figure 36.14. The program enables the user to select a locale, time zone, date style, and time style from the combo boxes.

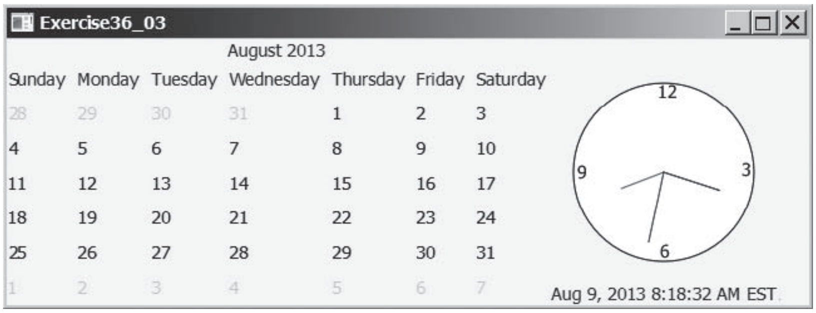


Date: venerdì, 9. agosto 2013 Time: 15.58 h EAT
Locale: Italian (Switzerland) it_CH Time Zone: Africa/Addis_Ababa
Date Style: Full Time Style: Full

FIGURE 36.14 The program displays the current date and time.

Section 36.3

36.3 (*Place the calendar and clock in a panel*) Write a program that displays the current date in a calendar and current time in a clock, as shown in Figure 36.15. Enable the program to run standalone.



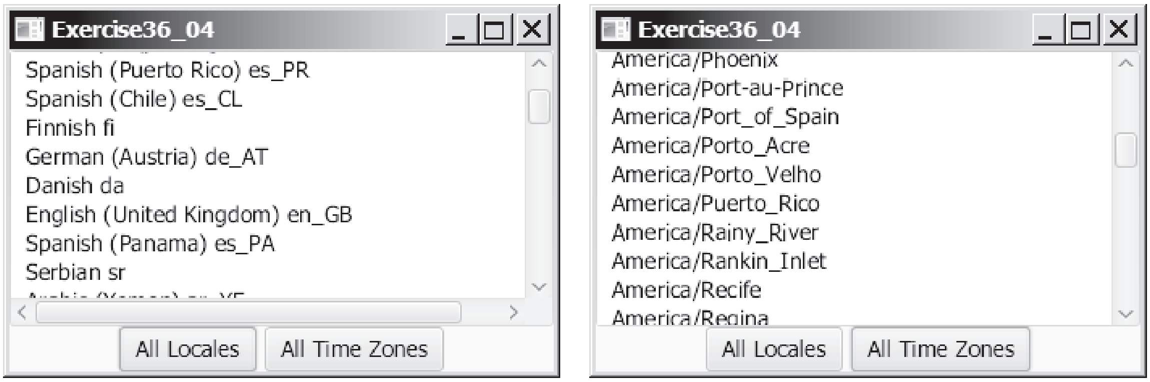
August 2013

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7

Aug 9, 2013 8:18:32 AM EST

FIGURE 36.15 The calendar and clock display the current date and time.

36.4 (*Find the available locales and time zone IDs*) Write two programs to display the available locales and time zone IDs using buttons, as shown in Figure 36.16.



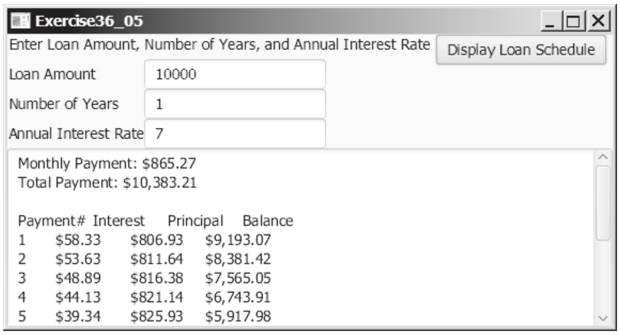
```
Spanish (Puerto Rico) es_PR
Spanish (Chile) es_CL
Finnish fi
German (Austria) de_AT
Danish da
English (United Kingdom) en_GB
Spanish (Panama) es_PA
Serbian sr
```

```
America/Phoenix
America/Port-au-Prince
America/Port_of_Spain
America/Porto_Acre
America/Porto_Velho
America/Puerto_Rico
America/Rainy_River
America/Rankin_Inlet
America/Recife
America/Regina
```

FIGURE 36.16 The program displays available locales and time zones using buttons.

Section 36.4

***36.5** (*Compute loan amortization schedule*) Rewrite Exercise 4.22 using JavaFX, as shown in Figure 36.17. The program allows the user to set the loan amount, loan



Enter Loan Amount, Number of Years, and Annual Interest Rate

Loan Amount 10000

Number of Years 1

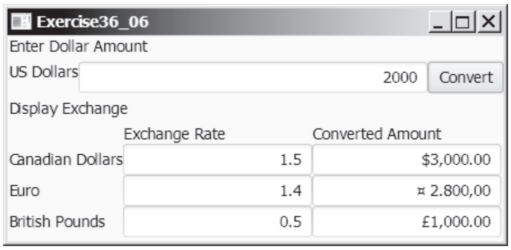
Annual Interest Rate 7

```
Monthly Payment: $865.27
Total Payment: $10,383.21

Payment# Interest   Principal   Balance
1        $58.33     $806.93     $9,193.07
2        $53.63     $811.64     $8,381.42
3        $48.89     $816.38     $7,565.05
4        $44.13     $821.14     $6,743.91
5        $39.34     $825.93     $5,917.98
```

FIGURE 36.17 The program displays the loan payment schedule.

period, and interest rate, and displays the corresponding interest, principal, and balance in the currency format.

36.6 (*Convert dollars to other currencies*) Write a program that converts U.S. dollars to Canadian dollars, German marks, and British pounds, as shown in Figure 36.18. The user enters the U.S. dollar amount and the conversion rate, and clicks the *Convert* button to display the converted amount.



Enter Dollar Amount

US Dollars 2000

Display Exchange

	Exchange Rate	Converted Amount
Canadian Dollars	1.5	$3,000.00
Euro	1.4	¤ 2.800,00
British Pounds	0.5	£1,000.00

FIGURE 36.18 The program converts U.S. dollars to Canadian dollars, German marks, and British pounds.

36.7 (*Compute loan payments*) Rewrite Listing 2.8, ComputeLoan.java, to display the monthly payment and total payment in currency.

36.8 (*Use the* `DecimalFormat` *class*) Rewrite Exercise 5.8 to display at most two digits after the decimal point for the temperature using the `DecimalFormat` class.

Section 36.5

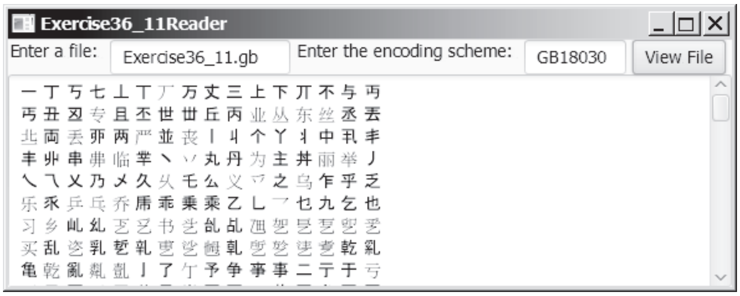
***36.9** (*Use resource bundle*) Modify the example for displaying a calendar in Section 36.3.6, "Example: Displaying a Calendar," to localize the labels "Choose a locale" and "Calendar Demo" in French, German, Chinese, or a language of your choice.

****36.10** (*Flag and anthem*) Rewrite Listing 16.13, ImageAudioAnimation.java, to use the resource bundle to retrieve image and audio files.

(*Hint*: When a new country is selected, set an appropriate locale for it. Have your program look for the flag and audio file from the resource file for the locale.)

Section 36.6

****36.11** (*Specify file encodings*) Write a program named `Exercise36_11Writer` that writes 1307 × 16 Chinese Unicode characters starting from `\u0E00` to a file named `Exercise36_11.gb` using the GBK encoding scheme. Output 16 characters per line and separate the characters with spaces. Write a program named `Exercise36_11Reader` that reads all the characters from a file using a specified encoding. Figure 36.19 displays the file using the GBK encoding scheme.



Enter a file: Exercise36_11.gb

Enter the encoding scheme: GB18030

```
一 丁 丂 七 丄 丅 丆 万 丈 三 上 下 丌 不 与 丏
丐 丑 丒 专 且 丕 世 丗 丘 丙 业 丛 东 丝 丞 丟
丠 両 丢 丣 两 严 並 丧 丨 丩 个 丫 丬 中 丮 丯
丰 丱 串 丳 临 丵 丶 丷 丸 丹 为 主 丼 丽 举 丿
乀 乁 乂 乃 乄 久 乆 乇 么 义 乊 之 乌 乍 乎 乏
乐 乑 乒 乓 乔 乕 乖 乗 乘 乙 乚 乛 乜 九 乞 也
习 乡 乢 乣 乤 乥 书 乧 乨 乩 乪 乫 乬 乭 乮 乯
买 乱 乲 乳 乴 乵 乶 乷 乸 乹 乺 乻 乼 乽 乾 乿
亀 亁 亂 亃 亄 亅 了 亇 予 争 亊 事 二 亍 于 亏
```

FIGURE 36.19 The program displays the file using the specified encoding scheme.

CHAPTER

37

SERVLETS

Objectives

- To explain how a servlet works (§37.2).
- To create/develop/run servlets (§37.3).
- To deploy servlets on application servers such as Tomcat and GlassFish (§37.3).
- To describe the servlets API (§37.4).
- To create simple servlets (§37.5).
- To create and process HTML forms (§37.6).
- To develop servlets to access databases (§37.7).
- To use hidden fields, cookies, and `HttpSession` to track sessions (§37.8).

37.1 Introduction

Key Point

Java Servlets is the foundation for developing Web applications using Java.

Servlets are Java programs that run on a Web server. They can be used to process client requests or produce dynamic webpages. For example, you can write servlets to generate dynamic webpages that display stock quotes or process client registration forms and store registration data in a database. This chapter introduces the concept of Java servlets. You will learn how to develop Java servlets using NetBeans.

Note

You can develop servlets without using an IDE. However, using an IDE such as NetBeans can greatly simplify the development task. The tool can automatically create the supporting directories and files. We choose NetBeans because it has the best support for Java Web development. You can still use your favorite IDE or no IDE for this chapter.

Note

Servlets are the foundation of Java Web technologies. JSP, JSF, and Java Web services are based on servlets. A good understanding of servlets helps you see the big picture of Java Web technology and learn JSP, JSF, and Web services.

37.2 HTML and Common Gateway Interface

Key Point

Java servlets are Java programs that function like CGI programs. They are executed upon request from a Web browser.

Java servlets run in the Web environment. To understand Java servlets, let us review HTML and the Common Gateway Interface (CGI).

37.2.1 Static Web Contents

You create webpages using HTML. Your webpages are stored as files on the Web server. The files are usually stored in the /htdocs directory on Unix, as shown in Figure 37.1. A user types a URL for the file from a Web browser. The browser contacts the Web server and requests the file. The server finds the file and returns it to the browser. The browser then displays the file to the user. This works fine for static information that does not change regardless of who requests it or when it is requested. Static information is stored in files. The information in the files can be updated, but at any given time every request for the same document returns exactly the same result.

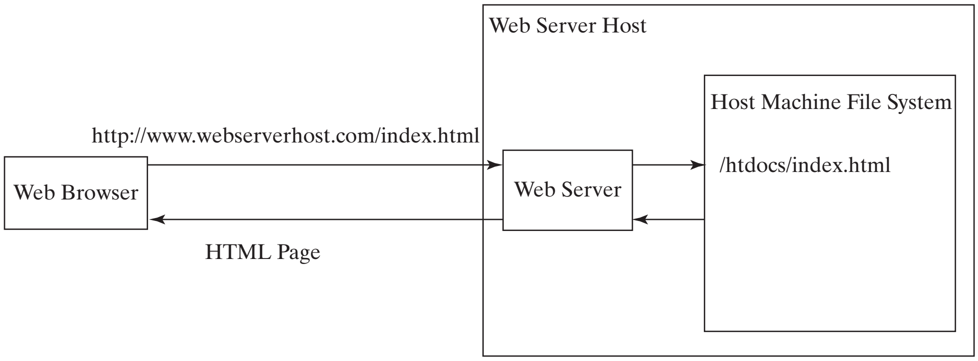


FIGURE 37.1 A Web browser requests a static HTML page from a Web server.

37.2.2 Dynamic Web Contents and Common Gateway Interface

Not all information, however, is static in nature. Stock quotes are updated whenever a trade takes place. Election vote counts are updated constantly on Election Day. Weather reports are frequently updated. The balance in a customer's bank account is updated whenever a transaction takes place. To view up-to-date information on the Web, the HTML pages for displaying this information must be generated dynamically. Dynamic Web pages are generated by Web servers. The Web server needs to run certain programs to process user requests from Web browsers in order to produce a customized response.

The *Common Gateway Interface*, or *CGI*, was proposed to generate dynamic Web content. The interface provides a standard framework for Web servers to interact with external programs, known as *CGI programs*. As shown in Figure 37.2, the Web server receives a request from a Web browser and passes it to the CGI program. The CGI program processes the request and generates a response at runtime. CGI programs can be written in any language, but the *Perl* language is the most popular choice. CGI programs are typically stored in the /cgi-bin directory. Here is a pseudocode example of a CGI program for displaying a customer's bank account balance:

1. Obtain account ID and password.
2. Verify account ID and password. If it fails, generate an HTML page to report incorrect account ID and password, and exit.
3. Retrieve account balance from the database; generate an HTML page to display the account ID and balance.

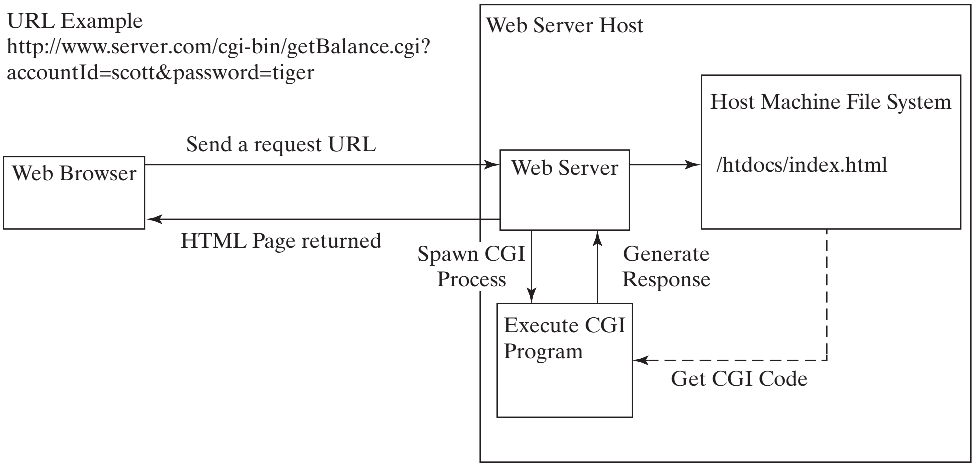

URL Example
http://www.server.com/cgi-bin/getBalance.cgi?
accountId=scott&password=tiger
Web Server Host
Host Machine File System
Send a request URL
Web Browser
Web Server
/htdocs/index.html
HTML Page returned
Spawn CGI Process
Generate Response
Execute CGI Program
Get CGI Code

FIGURE 37.2 A Web browser requests a dynamic HTML page from a Web server.

37.2.3 The GET and POST Methods

The two most common HTTP requests, also known as *methods*, are GET and POST. The Web browser issues a request using a URL or an HTML form to trigger the Web server to execute a CGI program. HTML forms will be introduced in §37.6, "HTML Forms." When issuing a CGI request directly from a URL, the GET method is used. This URL is known as a *query string*. The URL query string consists of the location of the CGI program, the parameters, and their values. For example, the following URL causes the CGI program `getBalance` to be invoked on the server side:

```
http://www.webserverhost.com/cgi-bin/
  getBalance.cgi?accountId=scott+smith&password=tiger
```

The `?` symbol separates the program from the parameters. The parameter name and value are associated using the = symbol. Parameter pairs are separated using the `&` symbol. The + symbol denotes a space character. So, here `accountId` is `scott smith`.

When issuing a request from an HTML form, either a GET method or a POST method can be used. The form explicitly specifies one of these. If the GET method is used, the data in the form are appended to the request string as if it were submitted using a URL. If the POST method is used, the data in the form are packaged as part of the request file. The server program obtains the data by reading the file. The POST method is more secure than the GET method.

Note

The GET and POST methods both send requests to the Web server. The POST method always triggers the execution of the corresponding CGI program. The GET method may not cause the CGI program to be executed, if the previous same request is cached in the Web browser. Web browsers often cache webpages so that the same request can be quickly responded to without contacting the Web server. The browser checks the request sent through the GET method as a URL query string. If the results for the exact same URL are cached on a disk, then the previous webpages for the URL may be displayed. To ensure that a new webpage is always displayed, use the POST method. For example, use a POST method if the request will actually update the database. If your request is not time sensitive, such as finding the address of a student in the database, use the GET method to speed up performance.

37.2.4 From CGI to Java Servlets

CGI provides a relatively simple approach for creating dynamic Web applications that accept a user request, process it on the server side, and return responses to the Web browser. But CGI is very slow when handling a large number of requests simultaneously, because the Web server spawns a process for executing each CGI program. Each process has its own runtime environment that contains and runs the CGI program. It is not difficult to imagine what will happen if many CGI programs were executed simultaneously. System resource would be quickly exhausted, potentially causing the server to crash.

Several new approaches have been developed to remedy the performance problem of CGI programs. Java servlets are one successful technology for this purpose. Java servlets are Java programs that function like CGI programs. They are executed upon request from a Web browser. All servlets run inside a *servlet container*, also referred to as a *servlet server* or a *servlet engine*. A servlet container is a single process that runs in a Java Virtual Machine. The JVM creates a thread to handle each servlet. Java threads have much less overhead than full-blown processes. All the threads share the same memory allocated to the JVM. Since the JVM persists beyond the life cycle of a single servlet execution, servlets can share objects already created in the JVM. For example, if multiple servlets access the same database, they can share the connection object. Servlets are much more efficient than CGI.

Servlets have other benefits that are inherent in Java. As Java programs, they are object oriented, portable, and platform independent. Since you know Java, you can develop servlets immediately with the support of Java API for accessing databases and network resources.

Check Point

37.2.1 What is the common gateway interface?

37.2.2 What are the differences between the GET and POST methods in an HTML form?

37.2.3 Can you submit a GET request directly from a URL? Can you submit a POST request directly from a URL?

37.2.4 What is wrong in the following URL for submitting a GET request to the servlet `FindScore` on host liang at port 8084 with parameter `name`?

http://liang:8084/findScore?name="P Yates"

37.2.5 What are the differences between CGI and servlets?

37.3 Creating and Running Servlets

An IDE such as NetBeans is an effective tool for creating Java servlet.

Key Point

To run Java servlets, you need a servlet container. Many servlet containers are available for free. Two popular ones are *Tomcat* (developed by Apache, www.apache.org) and *GlassFish* (developed by Sun, glassfish.dev.java.net). Both Tomcat and GlassFish are bundled and integrated with NetBeans 7 (Java EE version). When you run a servlet from NetBeans, Tomcat or GlassFish will be automatically started. You can choose to use either of them, or any other application server. GlassFish has more features than Tomcat and it takes more system resource.

37.3.1 Creating a Servlet

Before our introduction to the servlet API, let us look at a simple example to see how servlets work. A servlet to some extent resembles a JavaFX program. Every Java applet is a subclass of the `Application` class. You need to override appropriate methods in the `Application` class to implement the application. Every servlet is a subclass of the `HttpServlet` class. You need to override appropriate methods in the `HttpServlet` class to implement the servlet. Listing 37.1 is a servlet that generates a response in HTML using the `doGet` method.

LISTING 37.1 FirstServlet.java

```
 1  package chapter37;
 2
 3  import javax.servlet.*;
 4  import javax.servlet.http.*;
 5
 6  public class FirstServlet extends HttpServlet {
 7    /** Handle the HTTP GET method.
 8     * @param request servlet request
 9     * @param response servlet response
10     */
11    protected void doGet(HttpServletRequest request,
12        HttpServletResponse response)
13        throws ServletException, java.io.IOException {
14      response.setContentType("text/html");
15      java.io.PrintWriter out = response.getWriter();
16      // output your page here
17      out.println("<html>");
18      out.println("<head>");
19      out.println("<title>Servlet</title>");
20      out.println("</head>");
21      out.println("<body>");
22      out.println("Hello, Java Servlets");
23      out.println("</body>");
24      out.println("</html>");
25      out.close();
26    }
27  }
```

The `doGet` method (line 11) is invoked when the Web browser issues a request using the GET method. The `doGet` method has two parameters: `request` and `response`. `request` is for obtaining data from the Web browser, and `response` is for sending data back to the browser. Line 14 indicates that data are sent back to the browser as text/html. Line 15 obtains an instance of `PrintWriter` for actually outputting data to the browser.

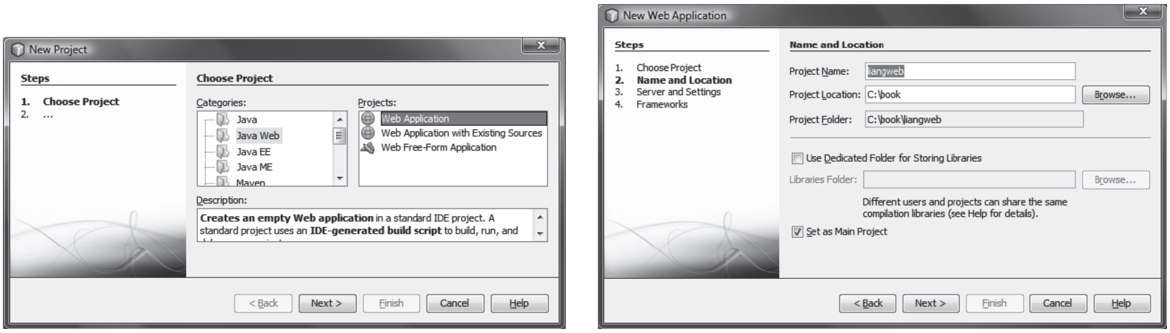
37.3.2 Creating Servlets in NetBeans

NetBeans is updated frequently. The current version is 8 at the time of this writing. To create a servlet in NetBeans 8, you have to first create a Web project, as follows:

1. Choose **File, New Project** to display the New Project dialog box. Choose **Java Web** in the Categories section and **Web Application** in the Projects section, as shown in Figure 37.3a. Click *Next* to display the New Web Application dialog box, as shown in Figure 37.3b.
2. Enter `liangweb` in the Project Name field and `c:\book` in the Project Location field. Check Set as Main Project. Click *Next* to display the dialog box for specifying server and settings, as shown in Figure 37.4.
3. Select `GlassFish Server 4.1` for server and `Java EE 7 Web` for Java EE Version. Click *Finish* to create the Web project, as shown in Figure 37.5.

Now you can create a servlet in the project, as follows:

1. Right-click the `liangweb` node in the project pane to display a context menu. Choose **New, Servlet** to display the New Servlet dialog box, as shown in Figure 37.6.


New Project
Steps
1. Choose Project
2. ...
Choose Project
Categories:
Java
Java Web
Java EE
Java ME
Maven
Projects:
Web Application
Web Application with Existing Sources
Web Free-Form Application
Description:
< Back
Next >
Finish
Cancel
Help
New Web Application
Steps
1. Choose Project
2. Name and Location
3. Server and Settings
4. Frameworks
Name and Location
Browse...
Libraries Folder:
Browse...
< Back
Next >
Finish
Cancel
Help

Creates an empty Web application in a standard IDE project. A standard project uses an **IDE-generated build script** to build, run, and

Project Name: liangweb

Project Location: C:\book

Project Folder: C:\book\liangweb

☐ Use Dedicated Folder for Storing Libraries

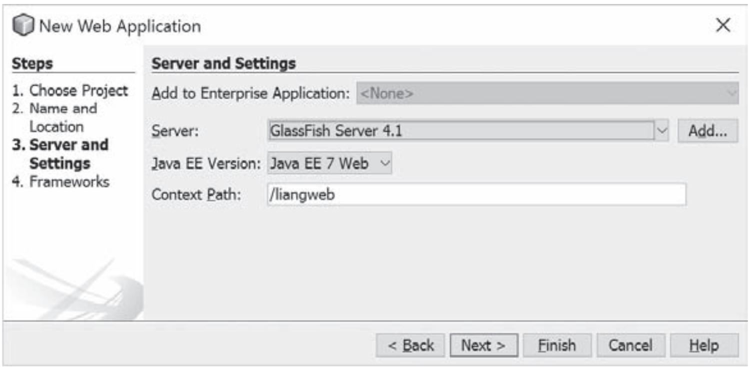
Different users and projects can share the same compilation libraries (see Help for details).

☑ Set as Main Project

(a)

(b)

FIGURE 37.3 (a) Choose Web Application to create a Web project. (b) Specify project name and location.


New Web Application
Steps
1. Choose Project
2. Name and Location
3. Server and Settings
4. Frameworks
Server and Settings
Add...
< Back
Next >
Finish
Cancel
Help

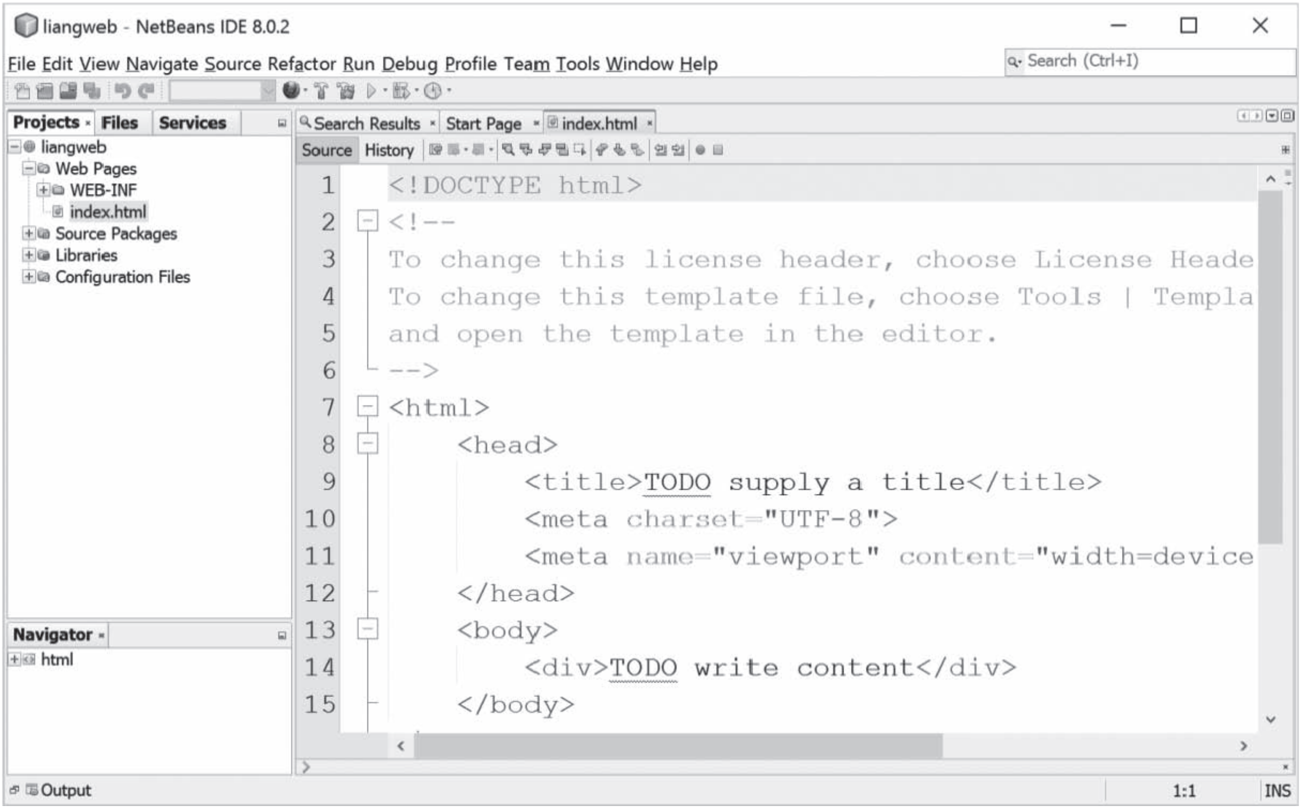
Add to Enterprise Application: <None>

Server: GlassFish Server 4.1

Java EE Version: Java EE 7 Web

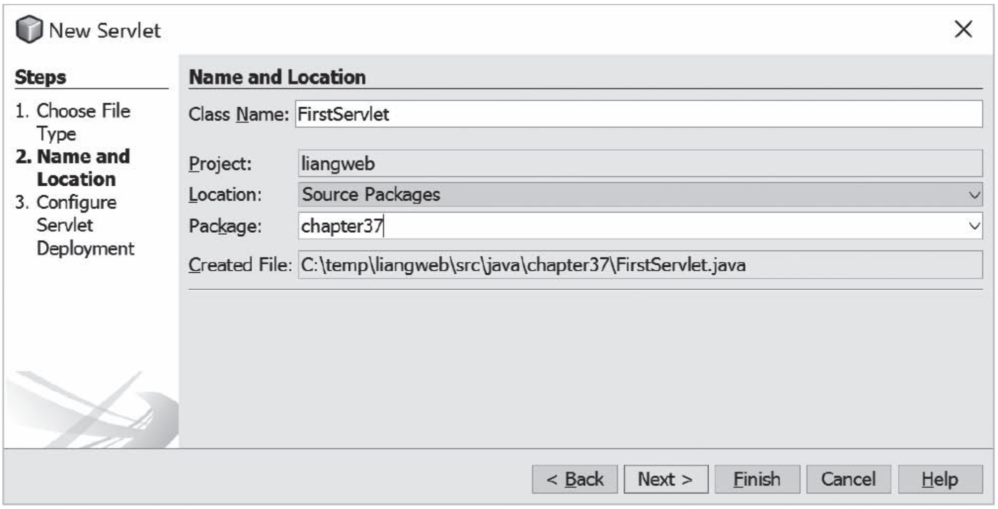
Context Path: /liangweb

FIGURE 37.4 Choose servers and settings.


liangweb - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
Projects
Files
Services
liangweb
Web Pages
WEB-INF
index.html
Source Packages
Libraries
Configuration Files
Navigator
html
Search Results
Start Page
index.html
Source
History
Output
1:1
INS

```
<!DOCTYPE html>
<!--
To change this license header, choose License Heade
To change this template file, choose Tools | Templa
and open the template in the editor.
-->
<html>
    <head>
        <title>TODO supply a title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device
    </head>
    <body>
        <div>TODO write content</div>
    </body>
```

FIGURE 37.5 A new Web project is created.


New Servlet
Steps
1. Choose File Type
2. Name and Location
3. Configure Servlet Deployment
< Back
Next >
Finish
Cancel
Help

Name and Location

Class Name: FirstServlet

Project: liangweb

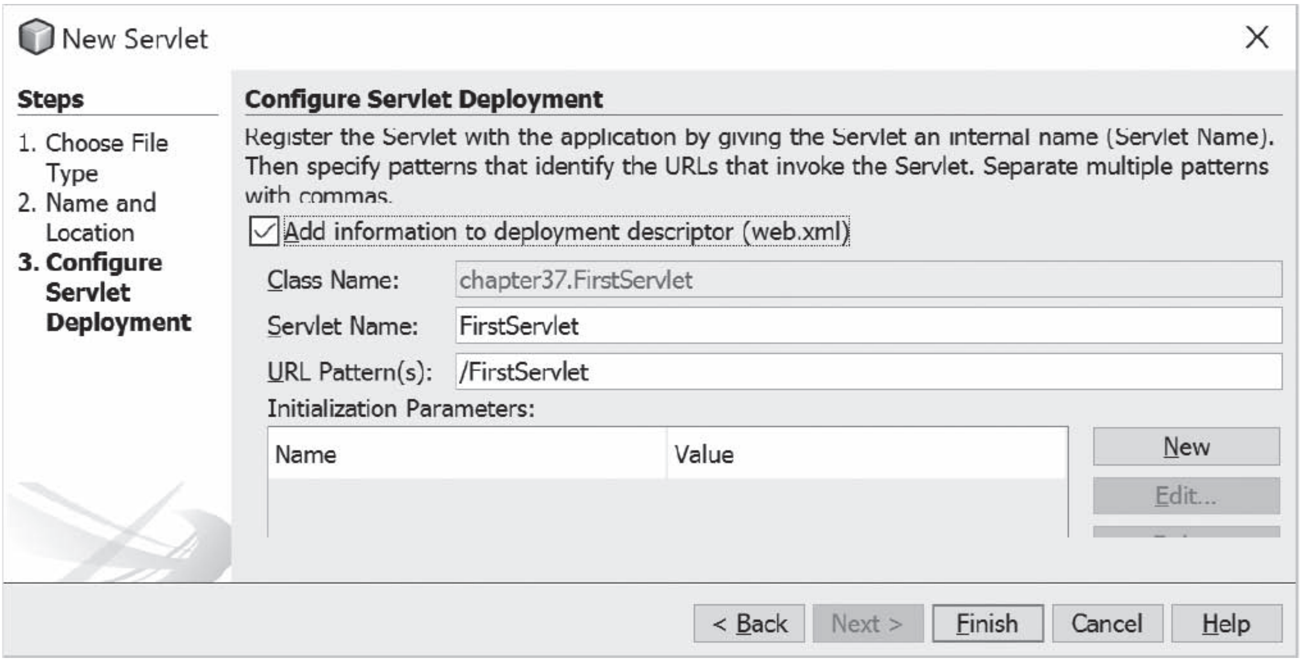
Location: Source Packages

Package: chapter37

Created File: C:\temp\liangweb\src\java\chapter37\FirstServlet.java

FIGURE 37.6 You can create a servlet in the New Servlet dialog box.

2. Enter `FirstServlet` in the Class Name field and chapter37 in the Package field and click *Next* to display the Configure Servlet Deployment dialog box, as shown in Figure 37.7.
3. Select the checkbox to add the servlet information to web.xml and click *Finish* to create the servlet. A servlet template is now created in the project, as shown in Figure 37.8.



New Servlet
Steps
1. Choose File Type
2. Name and Location
3. Configure Servlet Deployment
< Back
Next >
Finish
Cancel
Help
New
Edit...

Configure Servlet Deployment

Register the Servlet with the application by giving the Servlet an internal name (Servlet Name). Then specify patterns that identify the URLs that invoke the Servlet. Separate multiple patterns with commas.

☑ Add information to deployment descriptor (web.xml)

Class Name: chapter37.FirstServlet

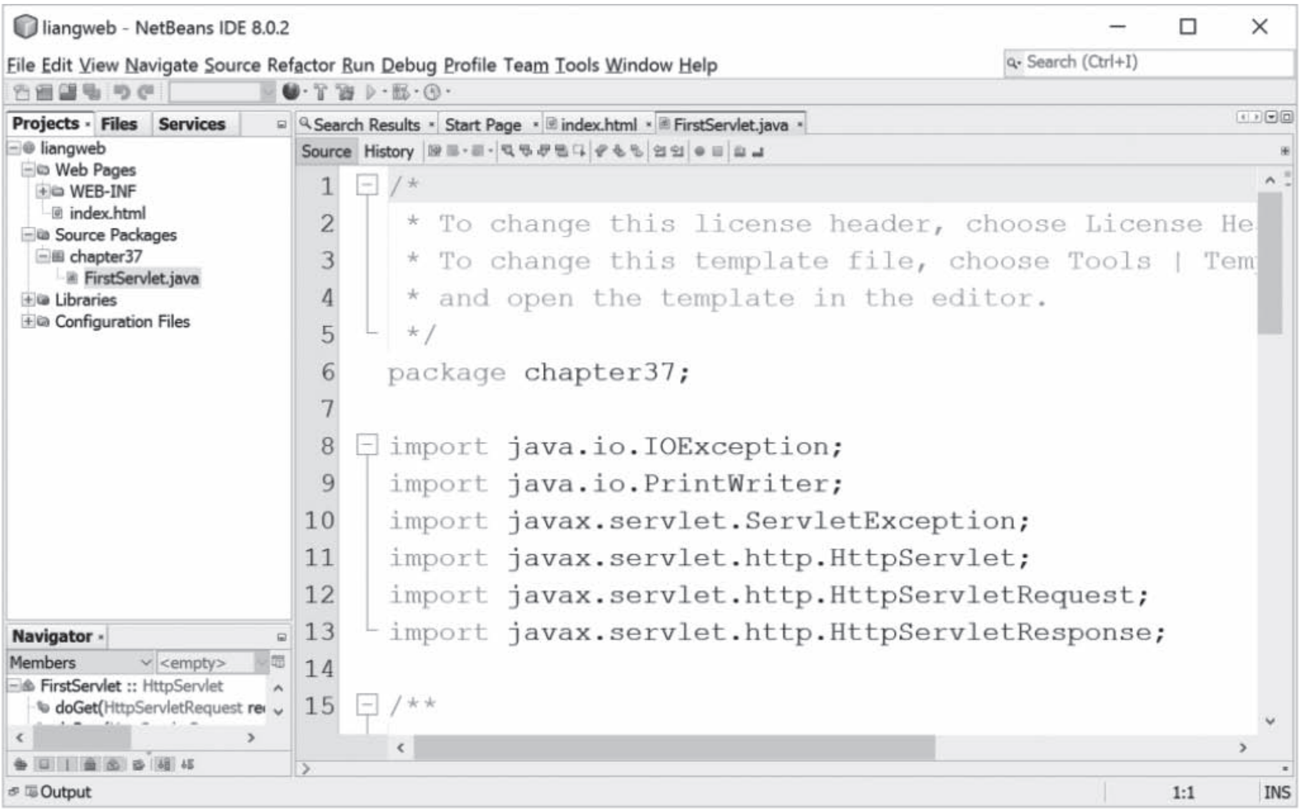
Servlet Name: FirstServlet

URL Pattern(s): /FirstServlet

Initialization Parameters:

Name	Value

FIGURE 37.7 You need to click the checkbox to add servlet information to `web.xml`.

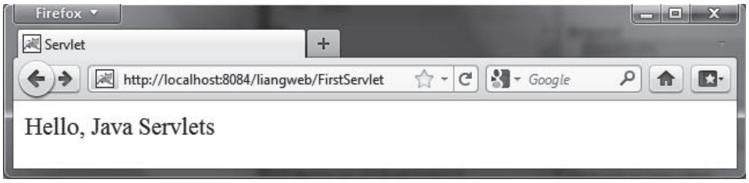


liangweb - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
Projects
Files
Services
liangweb
Web Pages
WEB-INF
index.html
Source Packages
chapter37
FirstServlet.java
Libraries
Configuration Files
Search Results
Start Page
index.html
FirstServlet.java
Source
History
Navigator
Members
<empty>
FirstServlet :: HttpServlet
doGet(HttpServletRequest re
Output
1:1
INS

```
/*
 * To change this license header, choose License He
 * To change this template file, choose Tools | Tem
 * and open the template in the editor.
 */
package chapter37;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
```

FIGURE 37.8 A new servlet class is created in the project.

4. Replace the code in the content pane for the servlet using the code in Listing 37.1.
5. Right-click `liangweb` node in the Project pane to display a context menu and choose **Run** to launch the Web server. In the Web browser, enter http://localhost:8084/liangweb/FirstServlet in the URL. You will now see the servlet result displayed, as shown in Figure 37.9.



Hello, Java Servlets

FIGURE 37.9 Servlet result is displayed in a Web browser.

Note

If the servlet is not displayed in the browser, do the following: 1. Make sure that you have added the servlet in the xml.web file. 2. Right-click `liangweb` and choose *Clean and Build*. 3. Right-click `liangweb` and choose *Run*. Reenter http://localhost:8084/liangweb/FirstServlet in the URL. If still not working, exit NetBeans and restart it.

Note

Depending on the server setup, you may have a port number other than **8084**.

Tip

You can deploy a Web application using a Web archive file (WAR) to a Web application server (e.g., Tomcat). To create a WAR file for the liangweb project, right-click liangweb and choose **Build Project**. You can now locate liangweb.war in the `c:\book\liangweb\dist` folder. To deploy on Tomcat, simply place liangweb.war into the webapps directory. When Tomcat starts, the .war file will be automatically installed.

Note

If you wish to use NetBeans as the development tool and Tomcat as the deployment server, please see Supplement V.E, "Tomcat Tutorial."

Check Point

37.3.1 Can you display an HTML file (e.g. `c:\ test.html`) by typing the complete file name in the Address field of Internet Explorer? Can you run a servlet by simply typing the servlet class file name?

37.3.2 How do you create a Web project in NetBeans?

37.3.3 How do you create a servlet in NetBeans?

37.3.4 How do you run a servlet in NetBeans?

37.3.5 When you run a servlet from NetBeans, what is the port number by default? What happens if the port number is already in use?

37.3.6 What is the .war file? How do you obtain a .war file for a Web project in NetBeans?

37.4 The Servlet API

Key Point

The `Servlet` *interface defines the methods* `init`, `service`, *and* `destroy` *to managing the life-cylce of a serlvet.*

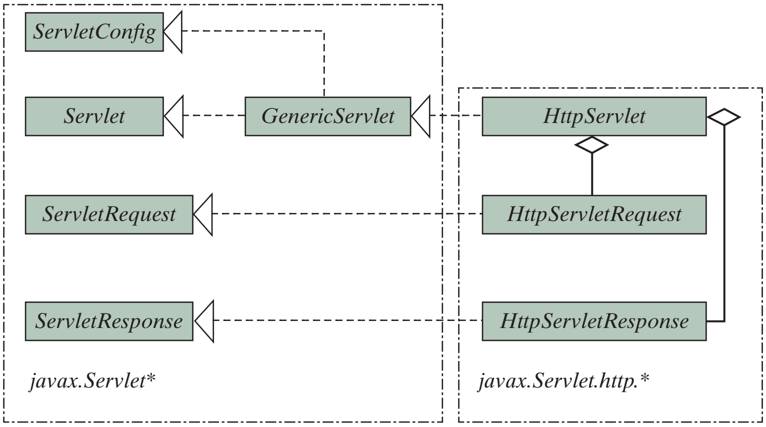

ServletConfig
Servlet
GenericServlet
HttpServlet
ServletRequest
HttpServletRequest
ServletResponse
HttpServletResponse
javax.Servlet*
javax.Servlet.http.*

FIGURE 37.10 The servlet API contains interfaces and classes that you use to develop and run servlets.

You have to know the servlet API in order to understand the source code in Listing3 7.1, in FirstServlet.java. The servlet API provides the interfaces and classes that support servlets. These interfaces and classes are grouped into two packages, `javax.servlet` and `javax.servlet.http`, as shown in Figure 37.10. The `javax.servlet` package provides basic interfaces, and the `javax.servlet.http` package provides classes and interfaces derived from them, which provide specific means for servicing HTTP requests.

37.4.1 The `Servlet` Interface

The `javax.servlet.Servlet` interface defines the methods that all servlets must implement. The methods are listed below:

```
/** Invoked for every servlet constructed */
public void init() throws ServletException;

/** Invoked to respond to incoming requests */
public void service(ServletRequest request, ServletResponse response)
  throws ServletException, IOException;

/** Invoked to release resource by the servlet */
public void destroy();
```

The `init`, `service`, and `destroy` methods are known as *life-cycle methods* and are called in the following sequence (see Figure 37.11):

1. The `init` method is called when the servlet is first created and is not called again as long as the servlet is not destroyed. This resembles an applet's `init` method, which is invoked after the applet is created and is not invoked again as long as the applet is not destroyed.

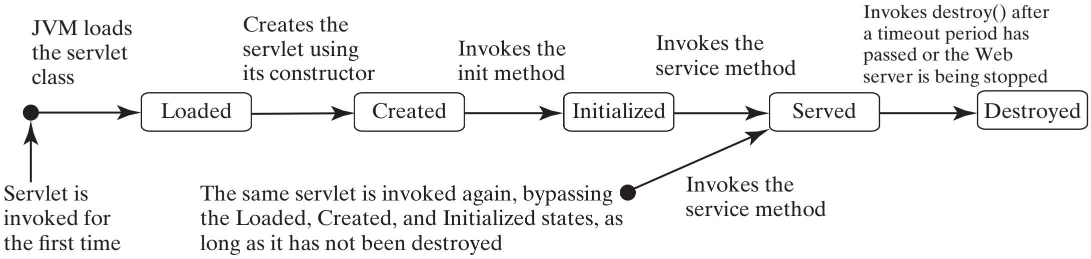

JVM loads the servlet class
Creates the servlet using its constructor
Invokes the init method
Invokes the service method
Invokes destroy() after a timeout period has passed or the Web server is being stopped
Loaded
Created
Initialized
Served
Destroyed
Servlet is invoked for the first time
The same servlet is invoked again, bypassing the Loaded, Created, and Initialized states, as long as it has not been destroyed
Invokes the service method

FIGURE 37.11 The JVM uses the `init`, `service`, and `destroy` methods to control the servlet.

2. The `service` method is invoked each time the server receives a request for the servlet. The server spawns a new thread and invokes `service`.
3. The `destroy` method is invoked after a timeout period has passed or as the Web server is terminated. This method releases resources for the servlet.

37.4.2 The `GenericServlet` Class, `ServletConfig` Interface, and `HttpServlet` Class

The `javax.servlet.GenericServlet` class defines a generic, protocol-independent servlet. It implements `javax.servlet.Servlet` and `javax.servlet.ServletConfig`. `ServletConfig` is an interface that defines four methods (`getInitParameter`, `getInitParameterNames`, `getServletContext`, and `getServletName`) for obtaining information from a Web server during initialization. All the methods in `Servlet` and `ServletConfig` are implemented in `GenericServlet` except `service`. Therefore, `GenericServlet` is an abstract class.

The `javax.servlet.http.HttpServlet` class defines a servlet for the HTTP protocol. It extends `GenericServlet` and implements the `service` method. The `service` method is implemented as a dispatcher of HTTP requests. The HTTP requests are processed in the following methods:

- `doGet` is invoked to respond to a GET request.
- `doPost` is invoked to respond to a POST request.
- `doDelete` is invoked to respond to a DELETE request. Such a request is normally used to delete a file on the server.
- `doPut` is invoked to respond to a PUT request. Such a request is normally used to send a file to the server.
- `doOptions` is invoked to respond to an OPTIONS request. This returns information about the server, such as which HTTP methods it supports.
- `doTrace` is invoked to respond to a TRACE request. Such a request is normally used for debugging. This method returns an HTML page that contains appropriate trace information.

All these methods use the following signature:

```
protected void doXxx(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, java.io.IOException
```

The `HttpServlet` class provides default implementation for these methods. You need to override `doGet`, `doPost`, `doDelete`, and `doPut` if you want the servlet to process a GET, POST, DELETE, or PUT request. By default, nothing will be done. Normally, you should not override the `doOptions` method unless the servlet implements new HTTP methods beyond those implemented by HTTP 1.1. Nor is there any need to override the `doTrace` method.

Note

GET and POST requests are often used, whereas DELETE, PUT, OPTIONS, and TRACE are not. For more information about these requests, please refer to the HTTP 1.1 specification from www.cis.ohio-state.edu/htbin/rfc/rfc2068.html.

Note

Although the methods in `HttpServlet` are all nonabstract, `HttpServlet` is defined as an abstract class. Thus you cannot create a servlet directly from `HttpServlet`. Instead you have to define your servlet by extending `HttpServlet`.

The relationship of these interfaces and classes is shown in Figure 37.12.

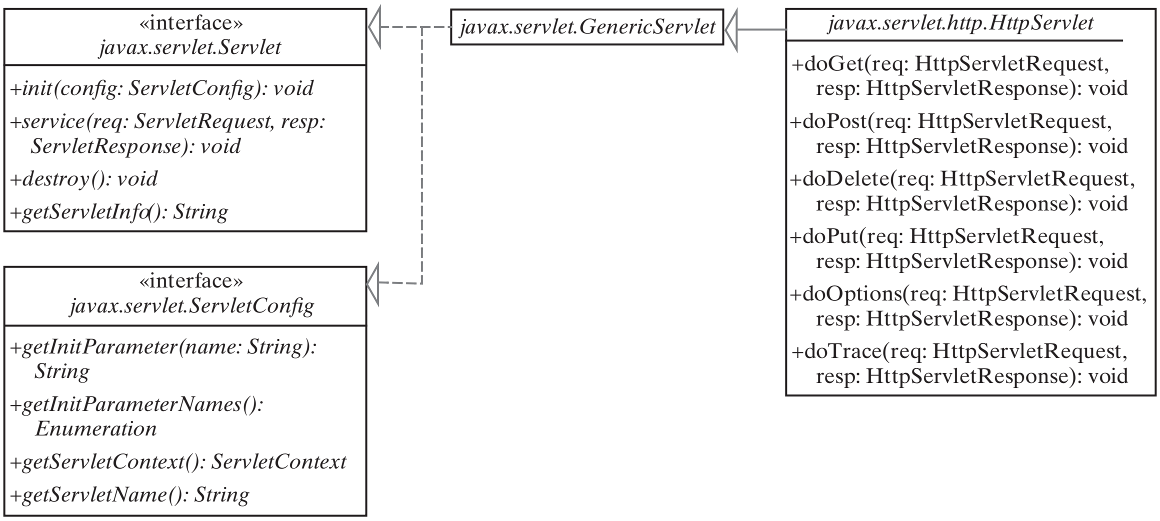

«interface»
javax.servlet.Servlet
+init(config: ServletConfig): void
+service(req: ServletRequest, resp: ServletResponse): void
+destroy(): void
+getServletInfo(): String
«interface»
javax.servlet.ServletConfig
+getInitParameter(name: String): String
+getInitParameterNames(): Enumeration
+getServletContext(): ServletContext
+getServletName(): String
javax.servlet.GenericServlet
javax.servlet.http.HttpServlet
+doGet(req: HttpServletRequest, resp: HttpServletResponse): void
+doPost(req: HttpServletRequest, resp: HttpServletResponse): void
+doDelete(req: HttpServletRequest, resp: HttpServletResponse): void
+doPut(req: HttpServletRequest, resp: HttpServletResponse): void
+doOptions(req: HttpServletRequest, resp: HttpServletResponse): void
+doTrace(req: HttpServletRequest, resp: HttpServletResponse): void

FIGURE 37.12 `HttpServlet` inherits abstract class `GenericServlet`, which implements interfaces `Servlet` and `ServletConfig`.

37.4.3 The `ServletRequest` Interface and `HttpServletRequest` Interface

Every `doXxx` method in the `HttpServlet` class has a parameter of the `HttpServletRequest` type, which is an object that contains HTTP request information, including parameter name and values, attributes, and an input stream. `HttpServletRequest` is a subinterface of `ServletRequest`. `ServletRequest` defines a more general interface to provide information for all kinds of clients. The frequently used methods in these two interfaces are shown in Figure 37.13.

37.4.4 The `ServletResponse` Interface and `HttpServletResponse` Interface

Every `doXxx` method in the `HttpServlet` class has a parameter of the `HttpServletResponse` type, which is an object that assists a servlet in sending a response to the client. `HttpServletResponse` is a subinterface of `ServletResponse`. `ServletResponse` defines a more general interface for sending output to the client.

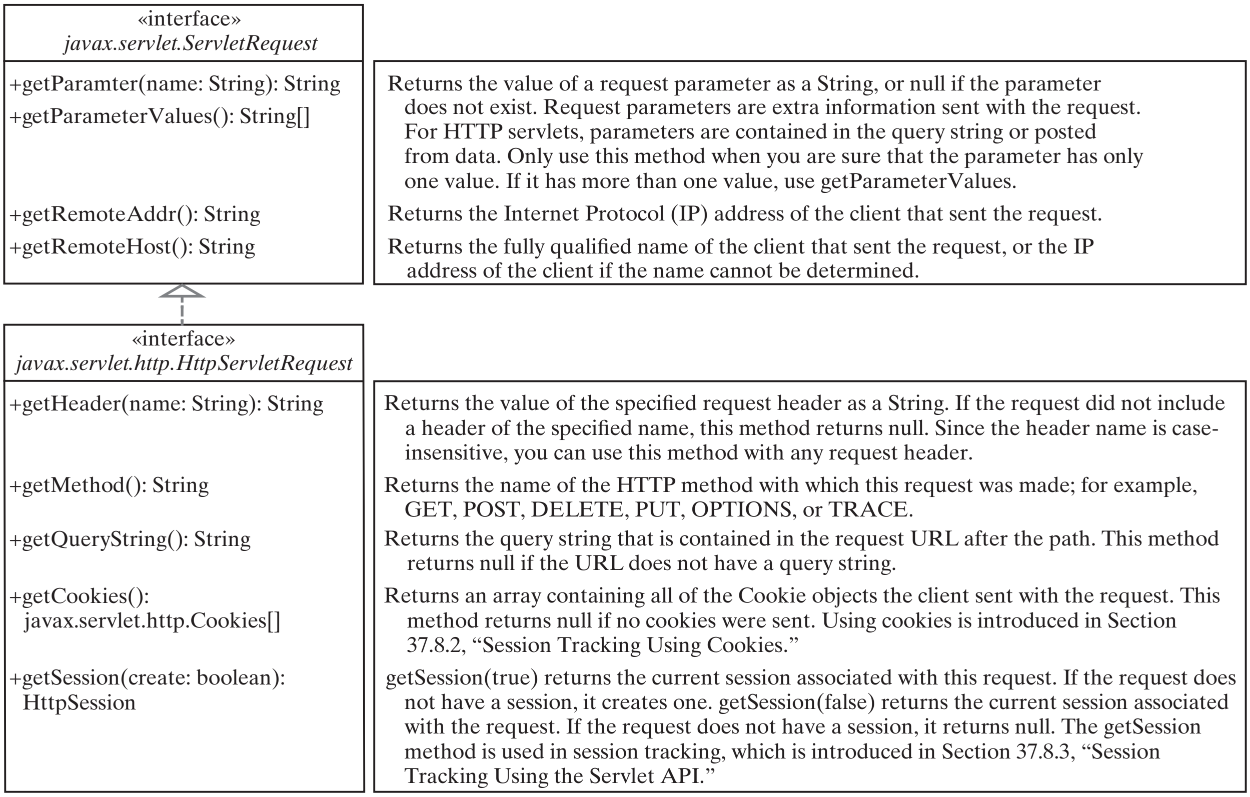
The frequently used methods in these two interfaces are shown in Figure 37.14.

Check Point

37.4.1 Describe the life cycle of a servlet.

37.4.2 Suppose you started the Web server, ran the following servlet twice by issuing an appropriate URL from the Web browser, and finally stopped Tomcat. What was displayed on the console when the servlet was first invoked? What was displayed on the console when the servlet was invoked for the second time? What was displayed on the console when Tomcat was shut down?

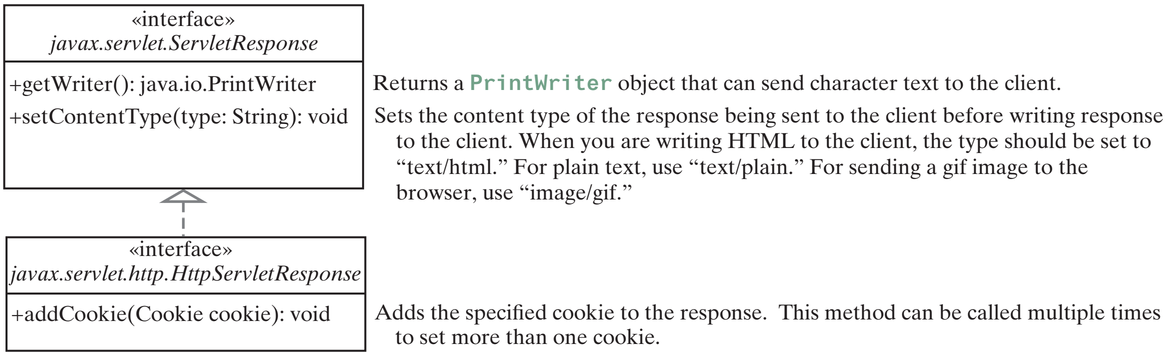
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class Test extends HttpServlet {
  public Test() {
    System.out.println("Constructor called");
  }

  /** Initialize variables */
  public void init() throws ServletException {
    System.out.println("init called");
  }

  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request,
      HttpServletResponse response)
        throws ServletException, IOException {
    System.out.println("doGet called");
  }

  /** Clean up resources */
  public void destroy() {
    System.out.println("destroy called");
  }
}
```



«interface» *javax.servlet.ServletRequest*	
+getParamter(name: String): String	Returns the value of a request parameter as a String, or null if the parameter does not exist. Request parameters are extra information sent with the request. For HTTP servlets, parameters are contained in the query string or posted from data. Only use this method when you are sure that the parameter has only one value. If it has more than one value, use getParameterValues.
+getParameterValues(): String[]	
+getRemoteAddr(): String	Returns the Internet Protocol (IP) address of the client that sent the request.
+getRemoteHost(): String	Returns the fully qualified name of the client that sent the request, or the IP address of the client if the name cannot be determined.

«interface» *javax.servlet.http.HttpServletRequest*	
+getHeader(name: String): String	Returns the value of the specified request header as a String. If the request did not include a header of the specified name, this method returns null. Since the header name is case-insensitive, you can use this method with any request header.
+getMethod(): String	Returns the name of the HTTP method with which this request was made; for example, GET, POST, DELETE, PUT, OPTIONS, or TRACE.
+getQueryString(): String	Returns the query string that is contained in the request URL after the path. This method returns null if the URL does not have a query string.
+getCookies(): javax.servlet.http.Cookies[]	Returns an array containing all of the Cookie objects the client sent with the request. This method returns null if no cookies were sent. Using cookies is introduced in Section 37.8.2, "Session Tracking Using Cookies."
+getSession(create: boolean): HttpSession	getSession(true) returns the current session associated with this request. If the request does not have a session, it creates one. getSession(false) returns the current session associated with the request. If the request does not have a session, it returns null. The getSession method is used in session tracking, which is introduced in Section 37.8.3, "Session Tracking Using the Servlet API."

FIGURE 37.13 `HttpServletRequest` is a subinterface of `ServletRequest`.



«interface» *javax.servlet.ServletResponse*	
+getWriter(): java.io.PrintWriter	Returns a `PrintWriter` object that can send character text to the client.
+setContentType(type: String): void	Sets the content type of the response being sent to the client before writing response to the client. When you are writing HTML to the client, the type should be set to "text/html." For plain text, use "text/plain." For sending a gif image to the browser, use "image/gif."

«interface» *javax.servlet.http.HttpServletResponse*	
+addCookie(Cookie cookie): void	Adds the specified cookie to the response. This method can be called multiple times to set more than one cookie.

FIGURE 37.14 `HttpServletResponse` is a subinterface of `ServletResponse`.

37.5 Creating Servlets

Key Point

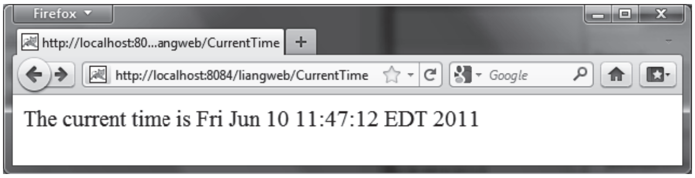
You can define a servlet class by extending the `HttpServlet` *class and implement the* `doGet` *and* `doPost` *methods.*

Servlets are the opposite of Java applets. Java applets run from a Web browser on the client side. To write Java programs, you define classes. To write a Java applet, you define a class that extends the `Applet` class. The Web browser runs and controls the execution of the applet through the methods defined in the `Applet` class. Similarly, to write a Java servlet, you define a class that extends the `HttpServlet` class. The servlet container runs and controls the execution of the servlet through the methods defined in the `HttpServlet` class. Like a Java applet, a servlet does not have a `main` method. A servlet depends on the servlet engine to call the methods. Every servlet has a structure like the one shown below:

```
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class MyServlet extends HttpServlet {
  /** Called by the servlet engine to initialize servlet */
  public void init() throws ServletException {
    ...
  }

  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    ...
  }

  /** Process the HTTP Post request */
  public void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    ...
  }

  /** Called by the servlet engine to release resource */
  public void destroy() {
    ...
  }

  // Other methods if necessary
}
```

The servlet engine controls the servlets using `init`, `doGet`, `doPost`, `destroy`, and other methods. By default, the `doGet` and `doPost` methods do nothing. To handle a GET request, you need to override the `doGet` method; to handle a POST request, you need to override the `doPost` method.

Listing 37.2 gives a simple Java servlet that generates a dynamic webpage for displaying the current time, as shown in Figure 37.15.


Firefox
http://localhost:80...angweb/CurrentTime
http://localhost:8084/liangweb/CurrentTime
Google

The current time is Fri Jun 10 11:47:12 EDT 2011

FIGURE 37.15 Servlet `CurrentTime` displays the current time.

LISTING 37.2 `CurrentTime.java`

```java
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class CurrentTime extends HttpServlet {
  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
      response)  throws ServletException, IOException {
      response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<p>The current time is " + new java.util.Date());
    out.close(); // Close stream
  }
}
```

The `HttpServlet` class has a `doGet` method. The `doGet` method is invoked when the browser issues a request to the servlet using the GET method. Your servlet class should override the `doGet` method to respond to the GET request. In this case, you write the code to display the current time.

Servlets return responses to the browser through an `HttpServletResponse` object. Since the `setContentType("text/html")` method sets the MIME type to “text/html,” the browser will display the response in HTML. The `getWriter` method returns a `PrintWriter` object for sending HTML back to the client.

Note

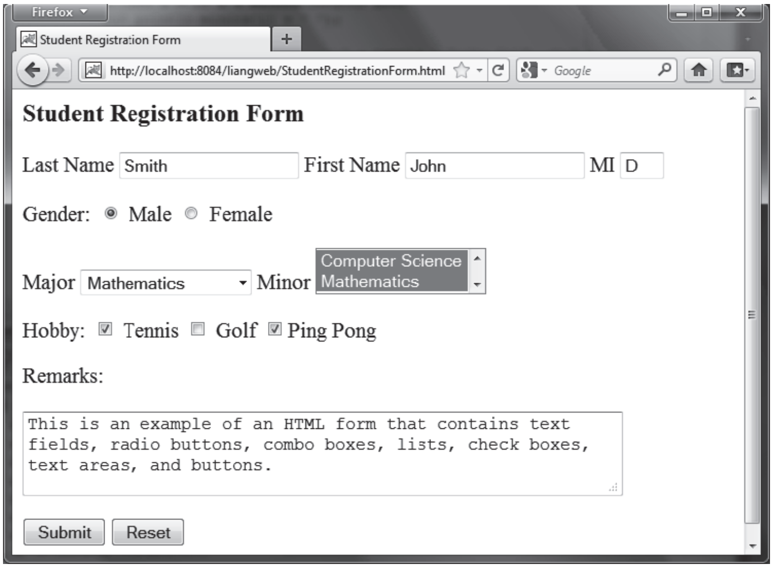
The URL query string uses the GET method to issue a request to the servlet. The current time may not be current if the webpage for displaying the current time is cached. To ensure that a new current time is displayed, refresh the page in the browser. In the next example, you will write a new servlet that uses the POST method to obtain the current time.

37.6 HTML Forms

Key Point

HTML forms are used to collect and submit data from a client to a Web server.

HTML forms enable you to submit data to the Web server in a convenient form. As shown in Figure 37.16, the form can contain text fields, text area, check boxes, combo boxes, lists, radio buttons, and buttons.


Firefox
Student Registration Form
http://localhost:8084/liangweb/StudentRegistrationForm.html
Google

Student Registration Form

Last Name Smith First Name John MI D

Gender: ● Male ○ Female

Major Mathematics Minor Computer Science, Mathematics

Hobby: ☑ Tennis ☐ Golf ☑ Ping Pong

Remarks:

```
This is an example of an HTML form that contains text
fields, radio buttons, combo boxes, lists, check boxes,
text areas, and buttons.
```

FIGURE 37.16 An HTML form may contain text fields, radio buttons, combo boxes, lists, check boxes, text areas, and buttons.

The HTML code for creating the form in Figure 37.16 is given in Listing 37.3. (If you are unfamiliar with HTML, please see Supplement V.A, “HTML and XHTML Tutorial.”)

LISTING 37.3 StudentRegistrationForm.html

```
<!--An HTML Form Demo -->
<html>
  <head>
    <title>Student Registration Form</title>
  </head>
  <body>
    <h3>Student Registration Form</h3>

    <form action = "GetParameters"
      method = "get">
      <!-- Name text fields -->
      <p><label>Last Name</label>
      <input type = "text" name = "lastName" size = "20" />
      <label>First Name</label>
      <input type = "text" name = "firstName" size = "20" />
      <label>MI</label>
      <input type = "text" name = "mi" size = "1" /></p>

      <!-- Gender radio buttons -->
      <p><label>Gender:</label>
      <input type = "radio" name = "gender" value = "M" checked />
      Male
      <input type = "radio" name = "gender" value = "F" /> Female</p>

      <!-- Major combo box -->
      <p><label>Major</label>
        <select name = "major" size = "1">
          <option value = "CS">Computer Science</option>
          <option value = "Math">Mathematics</option>
          <option>English</option>
          <option>Chinese</option>
        </select>

      <!-- Minor list -->
      <label>Minor</label>
        <select name = "minor" size = "2" multiple>
          <option>Computer Science</option>
          <option>Mathematics</option>
          <option>English</option>
          <option>Chinese</option>
        </select></p>

      <!-- Hobby check boxes -->
      <p><label>Hobby:</label>
        <input type = "checkbox" name = "tennis" /> Tennis
        <input type = "checkbox" name = "golf" /> Golf
        <input type = "checkbox" name = "pingPong" checked />Ping Pong
      </p>

      <!-- Remark text area -->
      <p>Remarks:</p>
      <p><textarea name = "remarks" rows = "3" cols = "56">
        </textarea></p>

```

```
   <!-- Submit and Reset buttons -->
   <p><input type = "submit" value = "Submit" />
   <input type = "reset" value = "Reset" /></p>
  </form>
 </body>
</html>
```

The following HTML tags are used to construct HTML forms:

- `<form>` . . . `</form>` defines a form body. The attributes for the `<form>` tag are `action` and `method`. The `action` attribute specifies the server program to be executed on the Web server when the form is submitted. The `method` attribute is either get or post.
- `<label>` . . . `</label>` simply defines a label.
- `<input>` defines an input field. The attributes for this tag are `type`, `name`, `value`, `checked`, `size`, and `maxlength`. The type attribute specifies the input type. Possible types are `text` for a one-line text field, `radio` for a radio button, and `checkbox` for a check box. The `name` attribute gives a formal name for the attribute. This `name` attribute is used by the servlet program to retrieve its associated value. The names of the radio buttons in a group must be identical. The value attribute specifies a default value for a text field and text area. The `checked` attribute indicates whether a radio button or a check box is initially checked. The `size` attribute specifies the size of a text field, and the `maxlength` attribute specifies the maximum length of a text field.
- `<select>` . . . `</select>` defines a combo box or a list. The attributes for this tag are `name`, `size`, and `multiple`. The `size` attribute specifies the number of rows visible in the list. The `multiple` attribute specifies that multiple values can be selected from a list. Set `size` to `1` and do not use a `multiple` for a combo box.
- `<option>` **. . .** `</option>` defines a selection list within a `<select>` . . . `</select>` tag. This tag may be used with the value attribute to specify a value for the selected option (e.g., `<option value = "CS">`Computer Science). If no value is specified, the selected option is the value.
- `<textarea>` . . . `</textarea>` defines a text area. The attributes are `name`, `rows`, and `cols`. The `rows` and `cols` attributes specify the number of rows and columns in a text area.

Note

You can create the HTML file from NetBeans. Right-click `liangweb` and choose *New, HTML*, to display the New HTML File dialog box. Enter `StudentRegistrationForm` as the file name and click *Finish* to create the file.

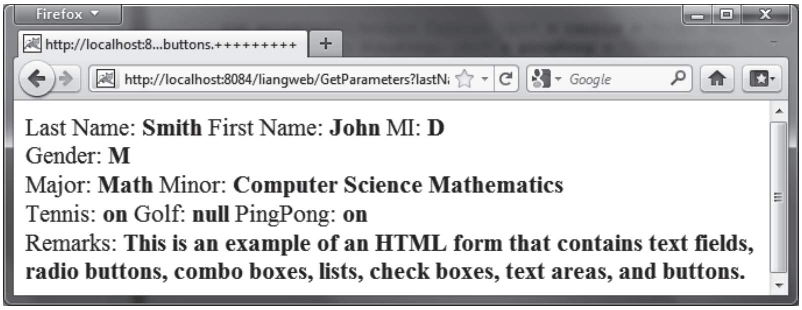
37.6.1 Obtaining Parameter Values from HTML Forms

To demonstrate how to obtain parameter values from an HTML form, Listing 37.4 creates a servlet to obtain all the parameter values from the preceding student registration form in Figure 37.16 and display their values, as shown in Figure 37.17.

LISTING 37.4 GetParameters.java

```
package chapter37;

import javax.servlet.*;
```

```
 4  import javax.servlet.http.*;
 5  import java.io.*;
 6
 7  public class GetParameters extends HttpServlet {
 8    /** Process the HTTP Post request */
 9    public void doGet(HttpServletRequest request, HttpServletResponse
10        response) throws ServletException, IOException {
11      response.setContentType("text/html");
12      PrintWriter out = response.getWriter();
13
14      // Obtain parameters from the client
15      String lastName = request.getParameter("lastName");
16      String firstName = request.getParameter("firstName");
17      String mi = request.getParameter("mi");
18      String gender = request.getParameter("gender");
19      String major = request.getParameter("major");
20      String[] minors = request.getParameterValues("minor");
21      String tennis = request.getParameter("tennis");
22      String golf = request.getParameter("golf");
23      String pingPong = request.getParameter("pingPong");
24      String remarks = request.getParameter("remarks");
25
26      out.println("Last Name: <b>" + lastName + "</b> First Name: <b>"
27        + firstName + "</b> MI: <b>" + mi + "</b><br>");
28      out.println("Gender: <b>" + gender + "</b><br>");
29      out.println("Major: <b>" + major + "</b> Minor: <b>");
30
31      if (minors != null)
32        for (int i = 0; i < minors.length; i++)
33          out.println(minors[i] + " ");
34
35      out.println("</b><br> Tennis: <b>" + tennis + "</b> Golf: <b>" +
36        golf + "</b> PingPong: <b>" + pingPong + "</b><br>");
37      out.println("Remarks: <b>" + remarks + "</b>");
38      out.close(); // Close stream
39    }
40  }
```


Firefox
http://localhost:8...buttons.+++++++++
http://localhost:8084/liangweb/GetParameters?lastN
Google

Last Name: **Smith** First Name: **John** MI: **D**
Gender: **M**
Major: **Math** Minor: **Computer Science Mathematics**
Tennis: **on** Golf: **null** PingPong: **on**
Remarks: **This is an example of an HTML form that contains text fields, radio buttons, combo boxes, lists, check boxes, text areas, and buttons.**

FIGURE 37.17 The servlet displays the parameter values entered in Figure 37.16.

The HTML form is already created in StudentRegistrationForm.html and displayed in Figure 37.16. Since the action for the form is `GetParameters`, clicking the *Submit* button invokes the `GetParameters` servlet.

Each GUI component in the form has a name attribute. The servlet uses the name attribute in the `getParameter(attributeName)` method to obtain the parameter value as a string. In case of a list with multiple values, use the `getParameterValues(attributeName)` method to return the parameter values in an array of strings (line 20).

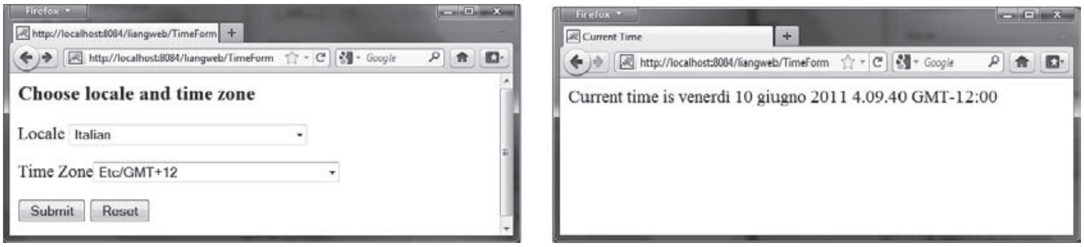
You may optionally specify the `value` attribute in a text field, text area, combo box, list, check box, or radio button in an HTML form. For text field and text area, the `value` attribute specifies a default value to be displayed in the text field and text area. The user can type in new values to replace it. For combo box, list, check box, and radio button, the `value` attribute specifies the parameter value to be returned from the `getParameter` and `getParameterValues` methods. If the `value` attribute is not specified for a combo box or a list, it returns the selected string from the combo box or the list. If the `value` attribute is not specified for a radio button or a check box, it returns string **on** for a checked radio button or a checked check box, and returns `null` for an unchecked check box.

Note

If an attribute does not exist, the `getParameter(attributeName)` method returns `null`. If an empty value of the parameter is passed to the servlet, the `getParameter(attributeName)` method returns a string with an empty value. In this case, the length of the string is `0`.

37.6.2 Obtaining Current Time Based on Locale and Time Zone

This example creates a servlet that processes the GET and POST requests. The GET request generates a form that contains a combo box for locale and a combo box for time zone, as shown in Figure 37.18a. The user can choose a locale and a time zone from this form to submit a POST request to obtain the current time based on the locale and time zone, as shown in Figure 37.18b.


Firefox
http://localhost:8084/liangweb/TimeForm
Google
Submit
Reset
Firefox
Current Time
http://localhost:8084/liangweb/TimeForm
Google

Choose locale and time zone

Locale: Italian

Time Zone: Etc/GMT+12

Current time is venerdi 10 giugno 2011 4.09.40 GMT-12:00

(a)

(b)

Figure 37.18 The GET method in the `TimeForm` servlet displays a form in (a), and the POST method in the `TimeForm` servlet displays the time based on locale and time zone in (b).

Listing 37.5 gives the servlet.

Listing 37.5 `TimeForm.java`

```java
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.text.*;

public class TimeForm extends HttpServlet {
  private static final String CONTENT_TYPE = "text/html";
  private Locale[] allLocale = Locale.getAvailableLocales();
  private String[] allTimeZone = TimeZone.getAvailableIDs();

  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
16        response) throws ServletException, IOException {
17      response.setContentType(CONTENT_TYPE);
18      PrintWriter out = response.getWriter();
19      out.println("<h3>Choose locale and time zone</h3>");
20      out.println("<form method=\"post\" action=" +
21        "TimeForm>");
22      out.println("Locale <select size=\"1\" name=\"locale\">");
23
24      // Fill in all locales
25      for (int i = 0; i < allLocale.length; i++) {
26        out.println("<option value=\"" + i +"\">" +
27          allLocale[i].getDisplayName() + "</option>");
28      }
29      out.println("</select>");
30
31      // Fill in all time zones
32      out.println("<p>Time Zone<select size=\"1\" name=\"timezone\">");
33      for (int i = 0; i < allTimeZone.length; i++) {
34        out.println("<option value=\"" + allTimeZone[i] +"\">" +
35          allTimeZone[i] + "</option>");
36      }
37      out.println("</select>");
38
39      out.println("<p><input type=\"submit\" value=\"Submit\" >");
40      out.println("<input type=\"reset\" value=\"Reset\"></p>");
41      out.println("</form>");
42      out.close(); // Close stream
43    }
44
45    /** Process the HTTP Post request */
46    public void doPost(HttpServletRequest request, HttpServletResponse
47        response) throws ServletException, IOException {
48      response.setContentType(CONTENT_TYPE);
49      PrintWriter out = response.getWriter();
50      out.println("<html>");
51      int localeIndex = Integer.parseInt(
52        request.getParameter("locale"));
53      String timeZoneID = request.getParameter("timezone");
54      out.println("<head><title>Current Time</title></head>");
55      out.println("<body>");
56      Calendar calendar =
57        new GregorianCalendar(allLocale[localeIndex]);
58      TimeZone timeZone = TimeZone.getTimeZone(timeZoneID);
59      DateFormat dateFormat = DateFormat.getDateTimeInstance(
60        DateFormat.FULL, DateFormat.FULL, allLocale[localeIndex]);
61      dateFormat.setTimeZone(timeZone);
62      out.println("Current time is " +
63        dateFormat.format(calendar.getTime()) + "</p>");
64      out.println("</body></html>");
65      out.close(); // Close stream
66    }
67  }
```

When you run this servlet, the servlet `TimeForm`'s `doGet` method is invoked to generate the time form dynamically. The method of the form is POST, and the action invokes the same servlet, `TimeForm`. When the form is submitted to the server, the `doPost` method is invoked to process the request.

The variables `allLocale` and `allTimeZone` (lines 11–12), respectively, hold all the available locales and time zone IDs. The names of the locales are displayed in the locale list. The values for the locales are the indexes of the locales in the array `allLocale`. The time zone IDs are strings. They are displayed in the time zone list. They are also the values for the list. The indexes of the locale and the time zone are passed to the servlet as parameters. The `doPost` method obtains the values of the parameters (lines 51–53) and finds the current time based on the locale and time zone.

Note

If you choose an Asian locale (e.g., Chinese, Korean, or Japanese), the time will not be displayed properly, because the default character encoding is UTF-8. To fix this problem, insert the following statement in line 48 to set an international character encoding:

```
response.setCharacterEncoding("GB18030");
```

For information on encoding, see Section 36.6.6, "Character Encoding."

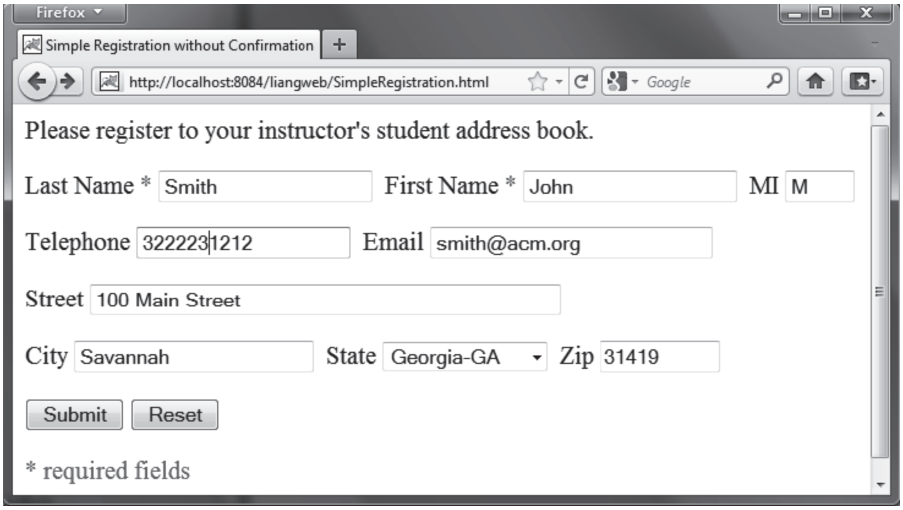
37.7 Database Programming in Servlets

Servlets can access and manipulate databases using JDBC.

Key Point

Many dynamic Web applications use databases to store and manage data. Servlets can connect to any relational database via JDBC. In Chapter 34, Java Database Programming, you learned how to create Java programs to access and manipulate relational databases via JDBC. Connecting a servlet to a database is no different from connecting a Java application or applet to a database. If you know Java servlets and JDBC, you can combine them to develop interesting and practical Web-based interactive projects.

To demonstrate connecting to a database from a servlet, let us create a servlet that processes a registration form. The client enters data in an HTML form and submits the form to the server, as shown in Figure 37.19. The result of the submission is shown in Figure 37.20. The server collects the data from the form and stores them in a database.

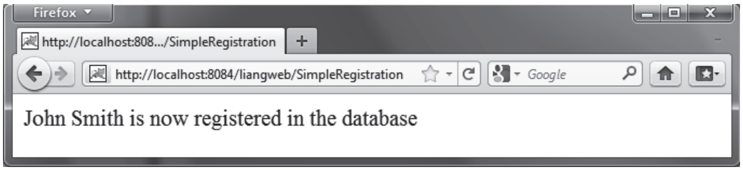

Firefox
Simple Registration without Confirmation
http://localhost:8084/liangweb/SimpleRegistration.html
Google
Submit
Reset

Please register to your instructor's student address book.

Last Name * Smith
First Name * John
MI M
Telephone 3222231212
Email smith@acm.org
Street 100 Main Street
City Savannah
State Georgia-GA
Zip 31419

* required fields

FIGURE 37.19 The HTML form enables the user to enter student information.


Firefox
http://localhost:808.../SimpleRegistration
http://localhost:8084/liangweb/SimpleRegistration
Google

John Smith is now registered in the database

FIGURE 37.20 The servlet processes the form and stores data in a database.

The registration data are stored in an `Address` table consisting of the following fields: `firstName`, `mi`, `lastName`, `street`, `city`, `state`, `zip`, `telephone`, and `email`, defined in the following statement:

```
create table Address (
  firstname varchar(25),
  mi char(1),
  lastname varchar(25),
  street varchar(40),
  city varchar(20),
  state varchar(2),
  zip varchar(5),
  telephone varchar(10),
  email varchar(30)
)
```

MySQL, Oracle, and Access were used in Chapter 34. You can use any relational database. If the servlet uses a database driver other than the JDBC-ODBC driver (e.g., the MySQL JDBC driver and the Oracle JDBC driver), you need to add the JDBC driver (e.g., mysqljdbc.jar for MySQL and ojdbc6.jar for Oracle) into the Libraries node in the project.

Create an HTML file named **SimpleRegistration.html** in Listing 37.6 for collecting the data and sending them to the database using the post method.

LISTING 37.6 `SimpleRegistration.html`

```
<!-- SimpleRegistration.html -->
<html>
  <head>
    <title>Simple Registration without Confirmation</title>
  </head>
  <body>
    Please register to your instructor's student address book.

    <form method = "post" action = "SimpleRegistration">
      <p>Last Name <font color = "#FF0000">*</font>
        <input type = "text" name = "lastName">&nbsp;
        First Name <font color = "#FF0000">*</font>
        <input type = "text" name = "firstName">&nbsp;
        MI <input type = "text" name = "mi" size = "3">
      </p>
      <p>Telephone
        <input type = "text" name = "telephone" size = "20">&nbsp;
        Email
        <input type = "text" name = "email" size = "28">&nbsp;
      </p>
      <p>Street <input type = "text" name = "street" size = "50">
      </p>
      <p>City <input type = "text" name = "city" size = "23">&nbsp;
        State
        <select size = "1" name = "state">
          <option value = "GA">Georgia-GA</option>
          <option value = "OK">Oklahoma-OK</option>
          <option value = "IN">Indiana-IN</option>
        </select>&nbsp;
        Zip <input type = "text" name = "zip" size = "9">
      </p>
      <p><input type = "submit" name = "Submit" value = "Submit">
         <input type = "reset" value = "Reset">
      </p>
    </form>
    <p><font color = "#FF0000">* required fields</font></p>
  </body>
</html>
```

Create the servlet named `SimpleRegistration` in Listing 37.7.

LISTING 37.7 SimpleRegistration.java

```
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class SimpleRegistration extends HttpServlet {
  // Use a prepared statement to store a student into the database
  private PreparedStatement pstmt;

  /** Initialize variables */
  public void init() throws ServletException {
    initializeJdbc();
  }

  /** Process the HTTP Post request */
  public void doPost(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Obtain parameters from the client
    String lastName = request.getParameter("lastName");
    String firstName = request.getParameter("firstName");
    String mi = request.getParameter("mi");
    String phone = request.getParameter("telephone");
    String email = request.getParameter("email");
    String address = request.getParameter("street");
    String city = request.getParameter("city");
    String state = request.getParameter("state");
    String zip = request.getParameter("zip");

    try {
      if (lastName.length() == 0 || firstName.length() == 0) {
        out.println("Last Name and First Name are required");
      }
      else {
        storeStudent(lastName, firstName, mi, phone, email,
          address, city, state, zip);

        out.println(firstName + " " + lastName +
          " is now registered in the database");
      }
    }
    catch(Exception ex) {
      out.println("Error: " + ex.getMessage());
    }
    finally {
      out.close(); // Close stream
    }
  }

54   /** Initialize database connection */
55   private void initializeJdbc() {
56     try {
57       // Load the JDBC driver
58       Class.forName("com.mysql.jdbc.Driver");
59       System.out.println("Driver loaded");
60
61       // Establish a connection
62       Connection conn = DriverManager.getConnection
63         ("jdbc:mysql://localhost/javabook", "scott", "tiger");
64       System.out.println("Database connected");
65
66       // Create a Statement
67       pstmt = conn.prepareStatement("insert into Address " +
68         "(lastName, firstName, mi, telephone, email, street, city, "
69         + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
70     }
71     catch (Exception ex) {
72       ex.printStackTrace();
73     }
74   }
75
76   /** Store a student record to the database */
77   private void storeStudent(String lastName, String firstName,
78       String mi, String phone, String email, String address,
79       String city, String state, String zip) throws SQLException {
80     pstmt.setString(1, lastName);
81     pstmt.setString(2, firstName);
82     pstmt.setString(3, mi);
83     pstmt.setString(4, phone);
84     pstmt.setString(5, email);
85     pstmt.setString(6, address);
86     pstmt.setString(7, city);
87     pstmt.setString(8, state);
88     pstmt.setString(9, zip);
89     pstmt.executeUpdate();
90   }
91 }
```

The `init` method (line 13) is executed once when the servlet starts. After the servlet has started, the servlet can be invoked many times as long as it is alive in the servlet container. Load the driver and connect to the database from the servlet's `init` method (line 14). If a prepared statement or a callable statement is used, it should also be created in the `init` method. In this example, a prepared statement is desirable, because the servlet always uses the same insert statement with different values.

A servlet can connect to any relational database via JDBC. The `initializeJdbc` method in this example connects to a MySQL database (line 58). Once connected, it creates a prepared statement for inserting a student record into the database. MySQL is used in this example; you can replace it with any relational database.

Last name and first name are required fields. If either of them is empty, the servlet sends an error message to the client (lines 35–36). Otherwise, the servlet stores the data in the database using the prepared statement.

Check Point

37.7.1 What would be displayed if you changed the content type to `html/plain` in Listing 37.2, CurrentTime.java?

37.7.2 The statement `out.close()` is used to close the output stream to response. Why isn't this statement enclosed in a try-catch block?

37.7.3 What happens when you invoke `request.getParameter(paramName)` if `paramName` does not exist?

37.7.4 How do you write a text field, combo box, check box, and text area in an HTML form?

37.7.5 How do you retrieve the parameter value for a text field, combo box, list, check box, radio button, and text area from an HTML form?

37.7.6 If the servlet uses a database driver other than the JDBC-ODBC driver, where should the driver be placed in NetBeans?

37.8 Session Tracking

You can perform session tracking using hidden values in a form, using cookies, or using `HttpSession`*.*

Key Point

Web servers use the Hyper-Text Transport Protocol (HTTP). HTTP is a stateless protocol. An HTTP Web server cannot associate requests from a client, and therefore treats each request independently. This protocol works fine for simple Web browsing, where each request typically results in an HTML file or a text file being sent back to the client. Such simple requests are isolated. However, the requests in interactive Web applications are often related. Consider the two requests in the following scenario:

> Request 1: A client sends registration data to the server; the server then returns the data to the user for confirmation.
>
> Request 2: The client confirms the data that was submitted in Request 1.

In Request 2, the data submitted in Request 1 are confirmed. These two requests are related in a session. A *session* can be defined as a series of related interactions between a single client and the Web server over a period of time. Tracking data among requests in a session is known as *session tracking*.

This section introduces three techniques for session tracking: using hidden values, using cookies, and using the session tracking tools from servlet API.

37.8.1 Session Tracking Using Hidden Values

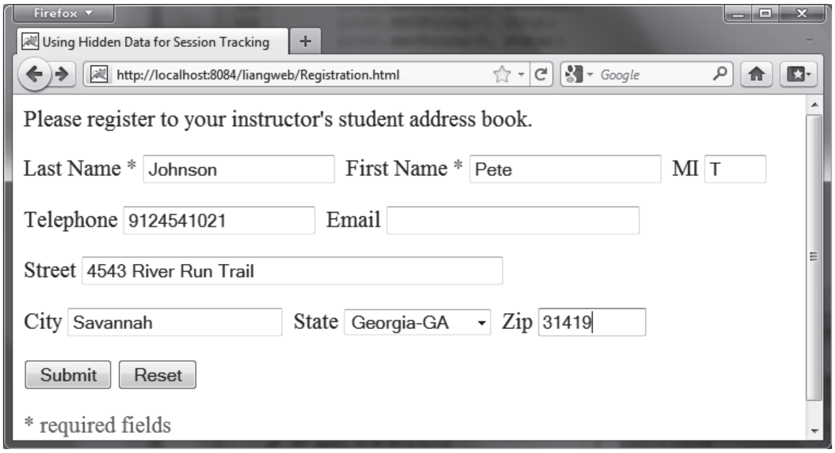
You can track a session by passing data from the servlet to the client as hidden values in a dynamically generated HTML form by including a field like this one:

```
<input type = "hidden" name = "lastName" value = "Smith">
```

The next request will submit the data back to the servlet. The servlet retrieves this hidden value just like any other parameter value, using the `getParameter` method.

Let us use an example to demonstrate using hidden values in a form. The example creates a servlet that processes a registration form. The client submits the form using the GET method, as shown in Figure 37.21. The server collects the data in the form, displays them to the client, and asks the client for confirmation, as shown in Figure 37.22. The client confirms the data by submitting the request with the hidden values using the POST method. Finally, the servlet writes the data to a database.

Create an HTML form named Registration.html in Listing 37.8 for collecting the data and sending it to the database using the GET method for confirmation. This file is almost identical to Listing 37.6, **SimpleRegistration.html** except that the action is replaced by `Registration` (line 9).


Firefox
Using Hidden Data for Session Tracking
http://localhost:8084/liangweb/Registration.html
Google
Submit
Reset

Please register to your instructor's student address book.

Last Name * Johnson First Name * Pete MI T

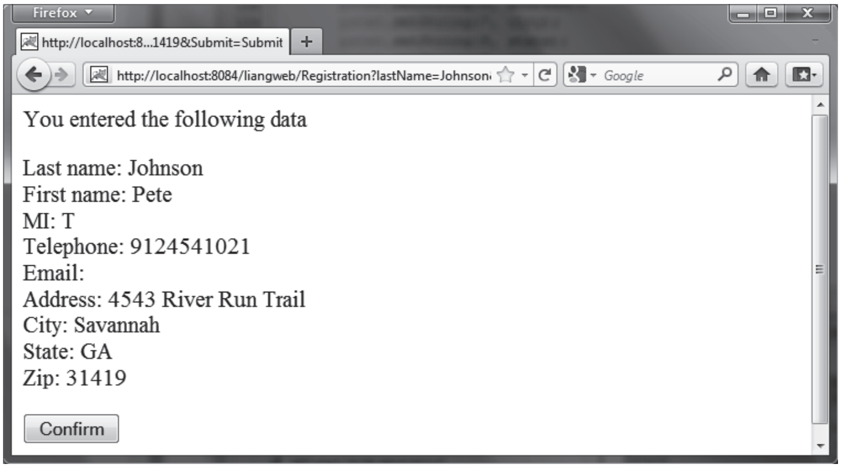
Telephone 9124541021 Email

Street 4543 River Run Trail

City Savannah State Georgia-GA Zip 31419

* required fields

FIGURE 37.21 The registration form collects user information.


Firefox
http://localhost:8...1419&Submit=Submit
http://localhost:8084/liangweb/Registration?lastName=Johnson
Google
Confirm

You entered the following data

Last name: Johnson
First name: Pete
MI: T
Telephone: 9124541021
Email:
Address: 4543 River Run Trail
City: Savannah
State: GA
Zip: 31419

FIGURE 37.22 The servlet asks the client for confirmation of the input.

LISTING 37.8 Registration.html

```
<!-- Registration.html -->
<html>
  <head>
    <title>Using Hidden Data for Session Tracking</title>
  </head>
  <body>
    Please register to your instructor’s student address book.

    <form method = "get" action = "Registration">
      <p>Last Name <font color = "#FF0000">*</font>
        <input type = "text" name = "lastName"> &nbsp;
        First Name <font color = "#FF0000">*</font>
        <input type = "text" name = "firstName"> &nbsp;
        MI  <input type = "text" name = "mi" size = "3">
      </p>
      <p>Telephone
        <input type = "text" name = "telephone" size = "20"> &nbsp;
        Email
```

```
          <input type = "text" name = "email" size = "28"> &nbsp;
        </p>
        <p>Street <input type = "text" name = "street" size = "50">
        </p>
        <p>City <input type = "text" name = "city" size = "23"> &nbsp;
          State
          <select size = "1" name = "state">
            <option value = "GA">Georgia-GA</option>
            <option value = "OK">Oklahoma-OK</option>
            <option value = "IN">Indiana-IN</option>
          </select> &nbsp;
          Zip <input type = "text" name = "zip" size = "9">
        </p>
        <p><input type = "submit" name = "Submit" value = "Submit">
           <input type = "reset" value = "Reset">
        </p>
      </form>
      <p><font color = "#FF0000">* required fields</font></p>
    </body>
  </html>
```

Create the servlet named Registration in Listing 37.9.

LISTING 37.9 Registration.java

```
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class Registration extends HttpServlet {
  // Use a prepared statement to store a student into the database
  private PreparedStatement pstmt;

  /** Initialize variables */
  public void init() throws ServletException {
    initializeJdbc();
  }

  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Obtain data from the form
    String lastName = request.getParameter("lastName");
    String firstName = request.getParameter("firstName");
    String mi = request.getParameter("mi");
    String telephone = request.getParameter("telephone");
    String email = request.getParameter("email");
    String street = request.getParameter("street");
    String city = request.getParameter("city");
    String state = request.getParameter("state");
    String zip = request.getParameter("zip");

    if (lastName.length() == 0 || firstName.length() == 0) {
      out.println("Last Name and First Name are required");
    }
```

```
      else {
        // Ask for confirmation
        out.println("You entered the following data");
        out.println("<p>Last name: " + lastName);
        out.println("<br>First name: " + firstName);
        out.println("<br>MI: " + mi);
        out.println("<br>Telephone: " + telephone);
        out.println("<br>Email: " + email);
        out.println("<br>Address: " + street);
        out.println("<br>City: " + city);
        out.println("<br>State: " + state);
        out.println("<br>Zip: " + zip);

        // Set the action for processing the answers
        out.println("<p><form method=\"post\" action=" +
          "Registration>");
        // Set hidden values
        out.println("<p><input type=\"hidden\" " +
          "value=" + lastName + " name=\"lastName\">");
        out.println("<p><input type=\"hidden\" " +
          "value=" + firstName + " name=\"firstName\">");
        out.println("<p><input type=\"hidden\" " +
          "value=" + mi + " name=\"mi\">");
        out.println("<p><input type=\"hidden\" " +
          "value=" + telephone + " name=\"telephone\">");
        out.println("<p><input type=\"hidden\" " +
          "value=" + email + " name=\"email\">");
        out.println("<p><input type=\"hidden\" " +
          "value=" + street + " name=\"street\">");
        out.println("<p><input type=\"hidden\" " +
          "value=" + city + " name=\"city\">");
        out.println("<p><input type=\"hidden\" " +
          "value=" + state + " name=\"state\">");
        out.println("<p><input type=\"hidden\" " +
          "value=" + zip + " name=\"zip\">");
        out.println("<p><input type=\"submit\" value=\"Confirm\" >");
        out.println("</form>");
      }

      out.close(); // Close stream
    }

    /** Process the HTTP Post request */
    public void doPost(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      try {
        String lastName = request.getParameter("lastName");
        String firstName = request.getParameter("firstName");
        String mi = request.getParameter("mi");
        String telephone = request.getParameter("telephone");
        String email = request.getParameter("email");
        String street = request.getParameter("street");
        String city = request.getParameter("city");
        String state = request.getParameter("state");
        String zip = request.getParameter("zip");

        storeStudent(lastName, firstName, mi, telephone, email,
          street, city, state, zip);
      out.println(firstName + " " + lastName +
        " is now registered in the database");
    }
    catch(Exception ex) {
      out.println("Error: " + ex.getMessage());
    }
  }

  /** Initialize database connection */
  private void initializeJdbc() {
    try {
      // Load the JDBC driver
      Class.forName("com.mysql.jdbc.Driver");
      System.out.println("Driver loaded");

      // Establish a connection
      Connection conn = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook" , "scott", "tiger");
      System.out.println("Database connected");

      // Create a Statement
      pstmt = conn.prepareStatement("insert into Address " +
        "(lastName, firstName, mi, telephone, email, street, city, "
        + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }

  /** Store a student record to the database */
  private void storeStudent(String lastName, String firstName,
      String mi, String phone, String email, String address,
      String city, String state, String zip) throws SQLException {
    pstmt.setString(1, lastName);
    pstmt.setString(2, firstName);
    pstmt.setString(3, mi);
    pstmt.setString(4, phone);
    pstmt.setString(5, email);
    pstmt.setString(6, address);
    pstmt.setString(7, city);
    pstmt.setString(8, state);
    pstmt.setString(9, zip);
    pstmt.executeUpdate();
  }
}
```

The servlet processes the GET request by generating an HTML page that displays the client's input and asks for the client's confirmation. The input data consist of hidden values in the newly generated forms, so they will be sent back in the confirmation request. The confirmation request uses the POST method. The servlet retrieves the hidden values and stores them in the database.

Since the first request does not write anything to the database, it is appropriate to use the GET method. Since the second request results in an update to the database, the POST method must be used.

Note

The hidden values could also be sent from the URL query string if the request used the GET method.

37.8.2 Session Tracking Using Cookies

You can track sessions using cookies, which are small text files that store sets of name/value pairs on the disk in the client's computer. Cookies are sent from the server through the instructions in the header of the HTTP response. The instructions tell the browser to create a cookie with a given name and its associated value. If the browser already has a cookie with the key name, the value will be updated. The browser will then send the cookie with any request submitted to the same server. Cookies can have expiration dates set, after which they will not be sent to the server. The `javax.servlet.http.Cookie` is used to create and manipulate cookies, as shown in Figure 37.23.

javax.servlet.http.Cookie	
+Cookie(name: String, value: String)	Creates a cookie with the specified name-value pair.
+getName(): String	Returns the name of the cookie.
+getValue(): String	Returns the value of the cookie.
+setValue(newValue: String): void	Assigns a new value to a cookie after the cookie is created.
+getMaxAge(): int	Returns the maximum age of the cookie, specified in seconds.
+setMaxAge(expiration: int): void	Specifies the maximum age of the cookie. By default, this value is –1, which implies that the cookie persists until the browser exits. If you set this value to 0, the cookie is deleted.
+getSecure(): boolean	Returns true if the browser is sending cookies only over a secure protocol.
+setSecure(flag: boolean): void	Indicates to the browser whether the cookie should only be sent using a secure protocol, such as HTTPS or SSL.
+getComment(): String	Returns the comment describing the purpose of this cookie, or null if the cookie has no comment.
+setComment(purpose: String): void	Sets the comment for this cookie.

FIGURE 37.23 `Cookie` stores a name/value pair and other information about the cookie.

To send a cookie to the browser, use the `addCookie` method in the `HttpServletResponse` class, as shown below:

```
response.addCookie(cookie);
```

where `response` is an instance of `HttpServletResponse`.

To obtain cookies from a browser, use

```
request.getCookies();
```

where `request` is an instance of `HttpServletRequest`.

To demonstrate the use of cookies, let us create an example that accomplishes the same task as Listing 37.9, Registration.java. Instead of using hidden values for session tracking, it uses cookies.

Create the servlet named RegistrationWithCookie in Listing 37.10. Create an HTML file named RegistrationWithCookie.html that is identical to Registration.html except that the action is replaced by `RegistrationWithCookie.java`.

LISTING 37.10 `RegistrationWithCookie.java`

```
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class RegistrationWithCookie extends HttpServlet {
  private static final String CONTENT_TYPE = "text/html";
  // Use a prepared statement to store a student into the database
  private PreparedStatement pstmt;

  /** Initialize variables */
  public void init() throws ServletException {
    initializeJdbc();
  }

  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Obtain data from the form
    String lastName = request.getParameter("lastName");
    String firstName = request.getParameter("firstName");
    String mi = request.getParameter("mi");
    String telephone = request.getParameter("telephone");
    String email = request.getParameter("email");
    String street = request.getParameter("street");
    String city = request.getParameter("city");
    String state = request.getParameter("state");
    String zip = request.getParameter("zip");

    if (lastName.length() == 0 || firstName.length() == 0) {
      out.println("Last Name and First Name are required");
    }
    else {
      // Create cookies and send cookies to browsers
      Cookie cookieLastName = new Cookie("lastName", lastName);
      // cookieLastName.setMaxAge(1000);
      response.addCookie(cookieLastName);
      Cookie cookieFirstName = new Cookie("firstName", firstName);
      response.addCookie(cookieFirstName);
      // cookieFirstName.setMaxAge(0);
      Cookie cookieMi = new Cookie("mi", mi);
      response.addCookie(cookieMi);
      Cookie cookieTelephone = new Cookie("telephone", telephone);
      response.addCookie(cookieTelephone);
      Cookie cookieEmail = new Cookie("email", email);
      response.addCookie(cookieEmail);
      Cookie cookieStreet = new Cookie("street", street);
      response.addCookie(cookieStreet);
      Cookie cookieCity = new Cookie("city", city);
      response.addCookie(cookieCity);
      Cookie cookieState = new Cookie("state", state);
      response.addCookie(cookieState);
      Cookie cookieZip = new Cookie("zip", zip);
      response.addCookie(cookieZip);

      // Ask for confirmation
      out.println("You entered the following data");
      out.println("<p>Last name: " + lastName);
      out.println("<br>First name: " + firstName);
      out.println("<br>MI: " + mi);
      out.println("<br>Telephone: " + telephone);
```

```
        out.println("<br>Email: " + email);
        out.println("<br>Street: " + street);
        out.println("<br>City: " + city);
        out.println("<br>State: " + state);
        out.println("<br>Zip: " + zip);

        // Set the action for processing the answers
        out.println("<p><form method=\"post\" action=" +
          "RegistrationWithCookie>");
        out.println("<p><input type=\"submit\" value=\"Confirm\" >");
        out.println("</form>");
      }

      out.close(); // Close stream
    }

    /** Process the HTTP Post request */
    public void doPost(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
      response.setContentType(CONTENT_TYPE);
      PrintWriter out = response.getWriter();

      String lastName = "";
      String firstName = "";
      String mi = "";
      String telephone = "";
      String email = "";
      String street = "";
      String city = "";
      String state = "";
      String zip = "";

      // Read the cookies
      Cookie[] cookies = request.getCookies();

      // Get cookie values
      for (int i = 0; i < cookies.length; i++) {
        if (cookies[i].getName().equals("lastName"))
          lastName = cookies[i].getValue();
        else if (cookies[i].getName().equals("firstName"))
          firstName = cookies[i].getValue();
        else if (cookies[i].getName().equals("mi"))
          mi = cookies[i].getValue();
        else if (cookies[i].getName().equals("telephone"))
          telephone = cookies[i].getValue();
        else if (cookies[i].getName().equals("email"))
          email = cookies[i].getValue();
        else if (cookies[i].getName().equals("street"))
          street = cookies[i].getValue();
        else if (cookies[i].getName().equals("city"))
          city = cookies[i].getValue();
        else if (cookies[i].getName().equals("state"))
          state = cookies[i].getValue();
        else if (cookies[i].getName().equals("zip"))
          zip = cookies[i].getValue();
      }

      try {
        storeStudent(lastName, firstName, mi, telephone, email, street,
          city, state, zip);

```

```
128          out.println(firstName + " " + lastName +
129            " is now registered in the database");
130
131          out.close(); // Close stream
132        }
133        catch(Exception ex) {
134          out.println("Error: " + ex.getMessage());
135        }
136      }
137
138      /** Initialize database connection */
139      private void initializeJdbc() {
140        try {
141          // Load the JDBC driver
142          Class.forName("com.mysql.jdbc.Driver");
143          System.out.println("Driver loaded");
144
145          // Establish a connection
146          Connection conn = DriverManager.getConnection
147            ("jdbc:mysql://localhost/javabook" , "scott", "tiger");
148          System.out.println("Database connected");
149
150          // Create a Statement
151          pstmt = conn.prepareStatement("insert into Address " +
152            "(lastName, firstName, mi, telephone, email, street, city, "
153            + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
154        }
155        catch (Exception ex) {
156          System.out.println(ex);
157        }
158      }
159
160      /** Store a student record to the database */
161      private void storeStudent(String lastName, String firstName,
162          String mi, String telephone, String email, String street,
163          String city, String state, String zip) throws SQLException {
164        pstmt.setString(1, lastName);
165        pstmt.setString(2, firstName);
166        pstmt.setString(3, mi);
167        pstmt.setString(4, telephone);
168        pstmt.setString(5, email);
169        pstmt.setString(6, street);
170        pstmt.setString(7, city);
171        pstmt.setString(8, state);
172        pstmt.setString(9, zip);
173        pstmt.executeUpdate();
174      }
175    }
```

You have to create a cookie for each value you want to track, using the `Cookie` class's only constructor, which defines a cookie's name and value as shown below (line 40):

```
Cookie cookieLastName = new Cookie("lastName", lastName);
```

To send the cookie to the browser, use a statement like this one (line 42):

```
response.addCookie(cookieLastName);
```

If a cookie with the same name already exists in the browser, its value is updated; otherwise, a new cookie is created.

Cookies are automatically sent to the Web server with each request from the client. The servlet retrieves all the cookies into an array using the `getCookies` method (line 100):

```
Cookie[] cookies = request.getCookies();
```

To obtain the name of the cookie, use the `getName` method (line 104):

```
String name = cookies[i].getName();
```

The cookie's value can be obtained using the `getValue` method:

```
String value = cookies[i].getValue();
```

Cookies are stored as strings just like form parameters and hidden values. If a cookie represents a numeric value, you have to convert it into an integer or a double, using the `parseInt` method in the `Integer` class or the `parseDouble` method in the `Double` class.

By default, a newly created cookie persists until the browser exits. However, you can set an expiration date, using the `setMaxAge` method, to allow a cookie to stay in the browser for up to 2,147,483,647 seconds (approximately 24,855 days).

37.8.3 Session Tracking Using the Servlet API

You have now learned both session tracking using hidden values and session tracking using cookies. These two session-tracking methods have problems. They send data to the browser either as hidden values or as cookies. The data are not secure, and anybody with knowledge of computers can obtain them. The hidden data are in HTML form, which can be viewed from the browser. Cookies are stored in the Cache directory of the browser. Because of security concerns, some browsers do not accept cookies. The client can turn the cookies off and limit their number. Another problem is that hidden data and cookies pass data as strings. You cannot pass objects using these two methods.

To address these problems, Java servlet API provides the `javax.servlet.http.HttpSession` interface, which provides a way to identify a user across more than one page request or visit to a website and to store information about that user. The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. A session usually corresponds to one user, who may visit a site many times. The session enables tracking of a large set of data. The data can be stored as objects and are secure because they are kept on the server side.

To use the Java servlet API for session tracking, first create a session object using the `getSession()` method in the `HttpServletRequest` interface:

```
HttpSession session = request.getSession();
```

This obtains the session or creates a new session if the client does not have a session on the server.

The `HttpSession` interface provides the methods for reading and storing data to the session, and for manipulating the session, as shown in Figure 37.24.

Note

HTTP is stateless. So how does the server associate a session with multiple requests from the same client? This is handled behind the scenes by the servlet container and is transparent to the servlet programmer.

To demonstrate using `HttpSession`, let us rewrite Listing 37.9, Registration.java, and Listing 37.10, RegistrationWithCookie.java. Instead of using hidden values or cookies for session tracking, it uses servlet `HttpSession`.

«interface» *javax.servlet.http.HttpSession*	
`+getAttribute(name: String): Object`	Returns the object bound with the specified name in this session, or null if no object is bound under the name.
`+setAttribute(name: String, value: Object): void`	Binds an object to this session, using the specified name. If an object of the same name is already bound to the session, the object is replaced.
`+getId(): String`	Returns a string containing the unique identifier assigned to this session. The identifier is assigned by the servlet container and is implementation dependent.
`+getLastAccessedTime(): long`	Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT, and marked by the time the container received the request.
`+invalidate(): void`	Invalidates this session, then unbinds any objects bound to it.
`+isNew(): boolean`	Returns true if the session was just created in the current request.
`+removeAttribute(name: String): void`	Removes the object bound with the specified name from this session. If the session does not have an object bound with the specified name, this method does nothing.
`+getMaxInactiveInterval(): int` `+setMaxInactiveInterval(interval: int): void`	Returns the time, in seconds, between client requests before the servlet container will invalidate this session. A negative time indicates that the session will never time-out. Use setMaxInactiveInterval to specify this value.

FIGURE 37.24 `HttpSession` establishes a persistent session between a client with multiple requests and the server.

Create the servlet named `RegistrationWithHttpSession` in Listing 37.11. Create an HTML file named `RegistrationWithHttpSession.html` that is identical to `Registration.html` except that the action is replaced by `RegistrationWithHttpSession`.

LISTING 37.11 `RegistrationWithHttpSession.java`

```java
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class RegistrationWithHttpSession extends HttpServlet {
  // Use a prepared statement to store a student into the database
  private PreparedStatement pstmt;

  /** Initialize variables */
  public void init() throws ServletException {
    initializeJdbc();
  }

  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
```

```
    // Set response type and output stream to the browser
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Obtain data from the form
    String lastName = request.getParameter("lastName");
    String firstName = request.getParameter("firstName");
    String mi = request.getParameter("mi");
    String telephone = request.getParameter("telephone");
    String email = request.getParameter("email");
    String street = request.getParameter("street");
    String city = request.getParameter("city");
    String state = request.getParameter("state");
    String zip = request.getParameter("zip");

    if (lastName.length() == 0 || firstName.length() == 0) {
      out.println("Last Name and First Name are required");
    }
    else {
      // Create an Address object
      Address address = new Address();
      address.setLastName(lastName);
      address.setFirstName(firstName);
      address.setMi(mi);
      address.setTelephone(telephone);
      address.setEmail(email);
      address.setStreet(street);
      address.setCity(city);
      address.setState(state);
      address.setZip(zip);

      // Get an HttpSession or create one if it does not exist
      HttpSession httpSession = request.getSession();

      // Store student object to the session
      httpSession.setAttribute("address", address);

      // Ask for confirmation
      out.println("You entered the following data");
      out.println("<p>Last name: " + lastName);
      out.println("<p>First name: " + firstName);
      out.println("<p>MI: " + mi);
      out.println("<p>Telephone: " + telephone);
      out.println("<p>Email: " + email);
      out.println("<p>Address: " + street);
      out.println("<p>City: " + city);
      out.println("<p>State: " + state);
      out.println("<p>Zip: " + zip);

      // Set the action for processing the answers
      out.println("<p><form method=\"post\" action=" +
        "RegistrationWithHttpSession>");
      out.println("<p><input type=\"submit\" value=\"Confirm\" >");
      out.println("</form>");
    }

    out.close(); // Close stream
  }

  /** Process the HTTP Post request */
```

```java
  public void doPost(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    // Set response type and output stream to the browser
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Obtain the HttpSession
    HttpSession httpSession = request.getSession();

    // Get the Address object in the HttpSession
    Address address = (Address)(httpSession.getAttribute("address"));

    try {
      storeStudent(address);

      out.println(address.getFirstName() + " " + address.getLastName()
        + " is now registered in the database");
      out.close(); // Close stream
    }
    catch(Exception ex) {
      out.println("Error: " + ex.getMessage());
    }
  }

  /** Initialize database connection */
  private void initializeJdbc() {
    try {
      // Load the JDBC driver
      Class.forName("com.mysql.jdbc.Driver");
      System.out.println("Driver loaded");

      // Establish a connection
      Connection conn = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook" , "scott", "tiger");
      System.out.println("Database connected");

      // Create a Statement
      pstmt = conn.prepareStatement("insert into Address " +
        "(lastName, firstName, mi, telephone, email, street, city, "
        + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }

  /** Store an address to the database */
  private void storeStudent(Address address) throws SQLException {
    pstmt.setString(1, address.getLastName());
    pstmt.setString(2, address.getFirstName());
    pstmt.setString(3, address.getMi());
    pstmt.setString(4, address.getTelephone());
    pstmt.setString(5, address.getEmail());
    pstmt.setString(6, address.getStreet());
    pstmt.setString(7, address.getCity());
    pstmt.setString(8, address.getState());
    pstmt.setString(9, address.getZip());
    pstmt.executeUpdate();
  }
}
```

The statement (line 52)

```
HttpSession httpSession = request.getSession();
```

obtains a session, or creates a new session if the session does not exist.

Since objects can be stored in `HttpSession`, this program defines an `Address` class. An `Address` object is created and is stored in the session using the `setAttribute` method, which binds the object with a name like the one shown below (line 55):

```
httpSession.setAttribute("address", address);
```

To retrieve the object, use the following statement (line 90):

```
Address address = (Address)(httpSession.getAttribute("address"));
```

There is only one session between a client and a servlet. You can store any number of objects in a session. By default, the maximum inactive interval on many Web servers including Tomcat and GlassFish is 1800 seconds (i.e., a half-hour), meaning that the session expires if there is no activity for 30 minutes. You can change the default using the `setMaxInactiveInterval` method. For example, to set the maximum inactive interval to one hour, use

```
httpSession.setMaxInactiveInterval(3600);
```

If you set a negative value, the session will never expire.

For this servlet program to work, you have to create the `Address` class in NetBeans, as follows:

1. Choose *New*, *Java Class* from the context menu of the `liangweb` node in the project pane to display the New Java Class dialog box.
2. Enter `Address` as the Class Name and `chapter37` as the package name. Click *Finish* to create the class.
3. Enter the code, as shown in Listing 37.12.

LISTING 37.12 Address.java

```
package chapter37;

public class Address {
private String firstName;
private String mi;
private String lastName;
private String telephone;
private String street;
private String city;
private String state;
private String email;
private String zip;

  public String getFirstName() {
    return this.firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getMi() {
    return this.mi;
  }
```

```

    public void setMi(String mi) {
      this.mi = mi;
    }

    public String getLastName() {
      return this.lastName;
    }

    public void setLastName(String lastName) {
      this.lastName = lastName;
    }

    public String getTelephone() {
      return this.telephone;
    }

    public void setTelephone(String telephone) {
      this.telephone = telephone;
    }

    public String getEmail() {
      return this.email;
    }

    public void setEmail(String email) {
      this.email = email;
    }

    public String getStreet() {
      return this.street;
    }

    public void setStreet(String street) {
      this.street = street;
    }

    public String getCity() {
      return this.city;
    }

    public void setCity(String city) {
      this.city = city;
    }

    public String getState() {
      return this.state;
    }

    public void setState(String state) {
      this.state = state;
    }

    public String getZip() {
      return this.zip;
    }

    public void setZip(String zip) {
      this.zip = zip;
    }
  }
```

This support class will also be reused in the upcoming chapters.

Check Point

37.8.1 What is session tracking? What are three techniques for session tracking?

37.8.2 How do you create a cookie, send a cookie to a browser, get cookies from a browser, get the name of a cookie, set a new value in the cookie, and set cookie expiration time?

37.8.3 Do you have to create five `Cookie` objects in the servlet in order to send five cookies to the browser?

37.8.4 How do you get a session, set object value for the session, and get object value from the session?

37.8.5 Suppose you inserted the following code in line 53 in Listing 37.11:

```
httpSession.setMaxInactiveInterval(1);
```

What would happen after the user clicked the *Confirm* button from the browser? Test your answer by running the program.

37.8.6 Suppose you inserted the following code in line 53 in Listing 37.11:

```
httpSession.setMaxInactiveInterval(-1);
```

What would happen after the user clicked the *Confirm* button from the browser?

KEY TERMS

Common Gateway Interface	37-3
CGI programs	37-3
cookie	37-30
GET and POST methods	37-3
GlassFish	37-5
HTML form	37-15
URL query string	37-3
servlet	37-2
servlet container (servlet engine)	37-4
servelt life-cycle methods	37-10
Tomcat	37-5

CHAPTER SUMMARY

1. A servlet is a special kind of program that runs from a Web server. Tomcat and GlassFish are Web servers that can run servlets.

2. A servlet URL is specified by the host name, port, and request string (e.g., http://localhost:8084/liangweb/ServletClass). There are several ways to invoke a servlet: (1) by typing a servlet URL from a Web browser, (2) by placing a hyper link in an HTML page, and (3) by embedding a servlet URL in an HTML form. All the requests trigger the GET method, except that in the HTML form you can explicitly specify the POST method.

3. You develop a servlet by defining a class that extends the `HttpServlet` class, implements the `doGet(HttpServletRequest, HttpServletResponse)` method to respond to the GET method, and implements the `doPost(HttpServletRequest, HttpServletResponse)` method to respond to the POST method.

4. The request information passed from a client to the servlet is contained in an object of `HttpServletRequest`. You can use the methods `getParameter`, `getParameterValues`, `getRemoteAddr`, `getRemoteHost`, `getHeader`, `getQueryString`, `getCookies`, and `getSession` to obtain the information from the request.

5. The content sent back to the client is contained in an object of `HttpServletResponse`. To send content to the client, first set the type of the content (e.g., html/plain) using the `setContentType(contentType)` method, then output the content through an I/O stream on the `HttpServletResponse` object. You can obtain a character `PrintWriter` stream using the `getWriter()` method and obtain a binary `OutputStream` using the `getOutputStream()` method.

6. A servlet may be shared by many clients. When the servlet is first created, its `init` method is called. It is not called again as long as the servlet is not destroyed. The `service` method is invoked each time the server receives a request for the servlet. The server spawns a new thread and invokes `service`. The `destroy` method is invoked after a timeout period has passed or the Web server is stopped.

7. There are three ways to track a session. You can track a session by passing data from the servlet to the client as a hidden value in a dynamically generated HTML form by including a field such as `<input type="hidden" name="lastName" value="Smith">`. The next request will submit the data back to the servlet. The servlet retrieves this hidden value just like any other parameter value using the `getParameter` method.

8. You can track sessions using cookies. A cookie is created using the constructor `new Cookie(String name, String value)`. Cookies are sent from the server through the object of `HttpServletResponse` using the `addCookie(aCookie)` method to tell the browser to add a cookie with a given name and its associated value. If the browser already has a cookie with the key name, the value will be updated. The browser will then send the cookie with any request submitted to the same server. Cookies can have expiration dates set, after which they will not be sent to the server.

9. Java servlet API provides a session-tracking tool that enables tracking of a large set of data. A session can be obtained using the `getSession()` method through an `HttpServletRequest` object. The data can be stored as objects and are secure because they are kept on the server side using the `setAttribute(String name, Object value)` method.

QUIZ

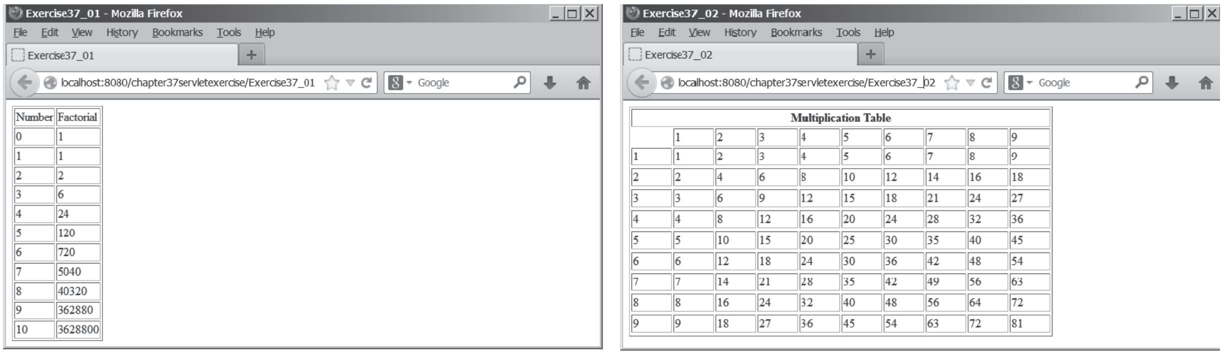
Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

Section 37.5

***37.1** (*Factorial table*) Write a servlet to display a table that contains factorials for the numbers from `0` to `10`, as shown in Figure 37.25.


Exercise37_01 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_01
localhost:8080/chapter37servletexercise/Exercise37_01
Google
Exercise37_02 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_02
localhost:8080/chapter37servletexercise/Exercise37_02
Google

Number	Factorial
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

Multiplication Table

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

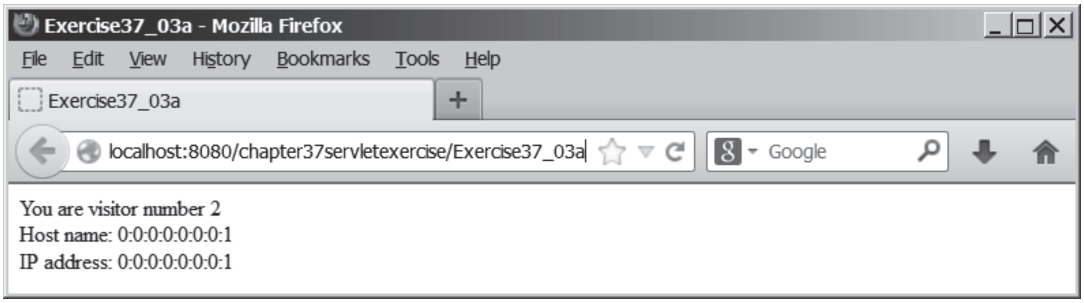
(a)

(b)

FIGURE 37.25 (a) The servlet displays factorials for the numbers from `0` to `10` in a table. (b) The servlet displays the multiplication table.

***37.2** (*Multiplication table*) Write a servlet to display a multiplication table, as shown in Figure 37.25b.

***37.3** (*Visit count*) Develop a servlet that displays the number of visits on the servlet. Also display the client's host name and IP address, as shown in Figure 37.26.


Exercise37_03a - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_03a
localhost:8080/chapter37servletexercise/Exercise37_03a
Google

```
You are visitor number 2
Host name: 0:0:0:0:0:0:0:1
IP address: 0:0:0:0:0:0:0:1
```

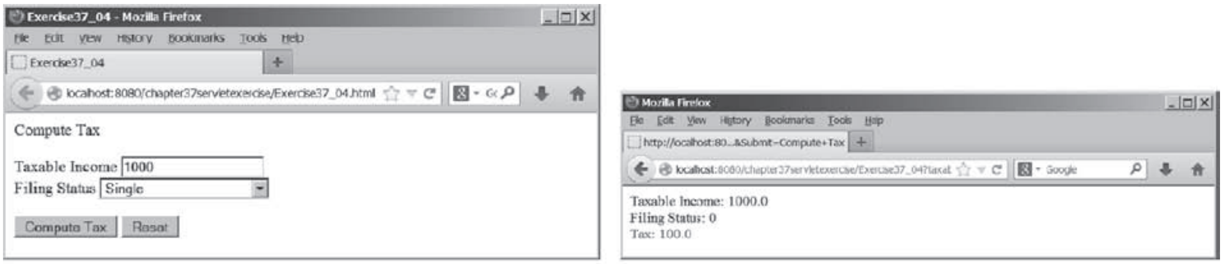
FIGURE 37.26 The servlet displays the number of visits and the client's host name, IP address, and request URL.

Implement this program in three different ways:

1. Use an instance variable to store `count`. When the servlet is created for the first time, `count` is `0`. `count` is incremented every time the servlet's `doGet` method is invoked. When the Web server stops, `count` is lost.
2. Store the count in a file named **Exercise39_3.dat**, and use `RandomAccessFile` to read the count in the servlet's `init` method. The count is incremented every time the servlet's `doGet` method is invoked. When the Web server stops, store the count back to the file.
3. Instead of counting total visits from all clients, count the visits by each client identified by the client's IP address. Use `Map` to store a pair of IP addresses and visit counts. For the first visit, an entry is created in the map. For subsequent visits, the visit count is updated.

Section 37.6

***37.4** (*Calculate tax*) Write an HTML form to prompt the user to enter taxable income and filing status, as shown in Figure 37.27a. Clicking the *Compute Tax* button invokes a servlet to compute and display the tax, as shown in Figure 37.27b. Use the `computeTax` method introduced in Listing 3.7, ComputingTax.java, to compute tax.


Exercise37_04 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_04
localhost:8080/chapter37servletexercise/Exercise37_04.html
Compute Tax Reset
Mozilla Firefox
http://localhost:80...&Submit=Compute+Tax
localhost:8080/chapter37servletexercise/Exercise37_04?taxat
Google

Compute Tax

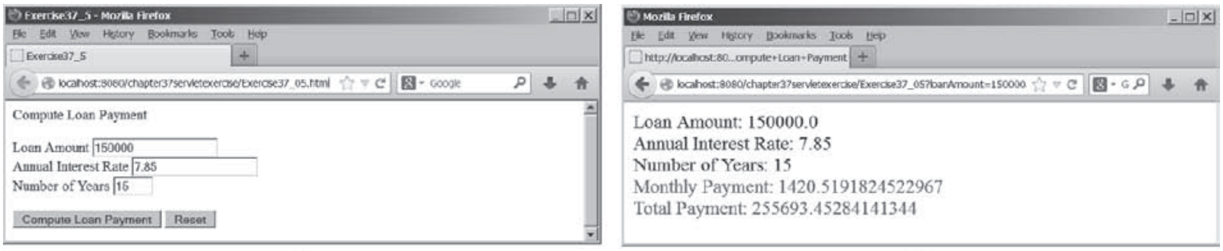
Taxable Income 1000
Filing Status Single

Taxable Income: 1000.0
Filing Status: 0
Tax: 100.0

(a) (b)

FIGURE 37.27 The servlet computes the tax.

***37.5** (*Calculate loan*) Write an HTML form that prompts the user to enter loan amount, interest rate, and number of years, as shown in Figure 37.28a. Clicking the *Compute Loan Payment* button invokes a servlet to compute and display the monthly and total loan payments, as shown in Figure 37.28b. Use the `Loan` class given in Listing 10.2, Loan.java, to compute the monthly and total payments.


Exercise37_5 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_5
localhost:8080/chapter37servletexercise/Exercise37_05.html
Google
Compute Loan Payment Reset
Mozilla Firefox
http://localhost:80...ompute+Loan+Payment
localhost:8080/chapter37servletexercise/Exercise37_05?loanAmount=150000

Compute Loan Payment

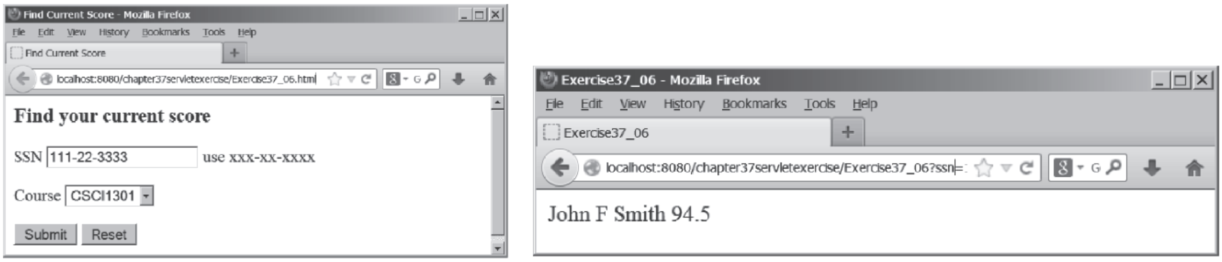
Loan Amount 150000
Annual Interest Rate 7.85
Number of Years 15

Loan Amount: 150000.0
Annual Interest Rate: 7.85
Number of Years: 15
Monthly Payment: 1420.5191824522967
Total Payment: 255693.45284141344

(a) (b)

FIGURE 37.28 The servlet computes the loan payment.

****37.6** (*Find scores from text files*) Write a servlet that displays the student name and the current score, given the SSN and class ID. For each class, a text file is used to store the student name, SSN, and current score. The file is named after the class ID with .txt extension. For instance, if the class ID were csci1301, the file name would be csci1301.txt. Suppose each line consists of student name, SSN, and score. These three items are separated by the # sign. Create an HTML form that enables the user to enter the SSN and class ID, as shown in Figure 37.29a. Upon clicking the *Submit* button, the result is displayed, as shown in Figure 37.29b. If the SSN or the class ID does not match, report an error. Assume three courses are available: CSCI1301, CSCI1302, and CSCI3720.


Find Current Score - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Find Current Score
localhost:8080/chapter37servletexercise/Exercise37_06.html
Submit Reset
Exercise37_06 - Mozilla Firefox
Exercise37_06
localhost:8080/chapter37servletexercise/Exercise37_06?ssn=

Find your current score

SSN 111-22-3333 use xxx-xx-xxxx

Course CSCI1301

John F Smith 94.5

(a) (b)

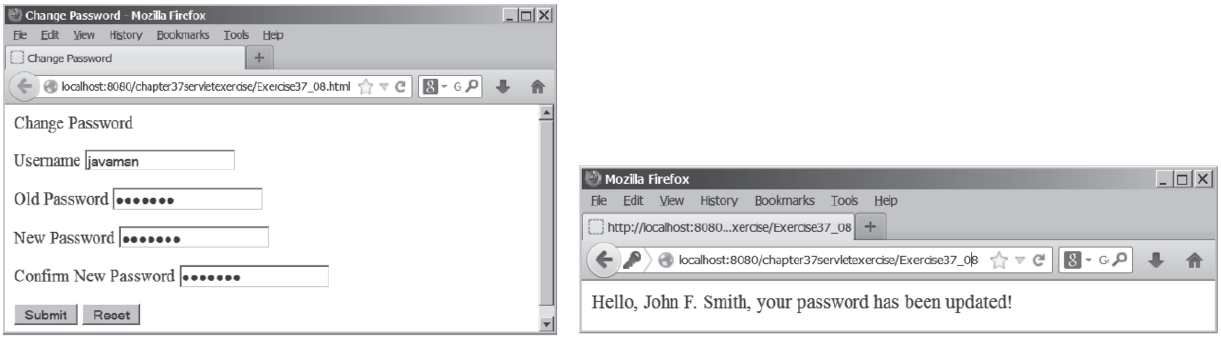
FIGURE 37.29 The HTML form accepts the SSN and class ID from the user and sends them to the servlet to obtain the score.

Section 37.7

****37.7** (*Find scores from database tables*) Rewrite the preceding servlet. Assume for each class, a table is used to store the student name, ssn, and score. The table name is the same as the class ID. For instance, if the class ID were csci1301, the table name would be csci1301.

***37.8** (*Change the password*) Write a servlet that enables the user to change the password from an HTML form, as shown in Figure 37.30a. Suppose the user information is stored in a database table named Account with three columns: username, password, and name, where name is the real name of the user. The servlet performs the following tasks:

a. Verify that the username and old password are in the table. If not, report the error and redisplay the HTML form.
b. Verify that the new password and the confirmed password are the same. If not, report this error and redisplay the HTML form.
c. If the user information is entered correctly, update the password and report the status of the update to the user, as shown in Figure 37.30b.


Change Password - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Change Password
localhost:8080/chapter37servletexercise/Exercise37_08.html
Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080...xercise/Exercise37_08
localhost:8080/chapter37servletexercise/Exercise37_08

Change Password

Username javamen

Old Password •••••••

New Password •••••••

Confirm New Password •••••••

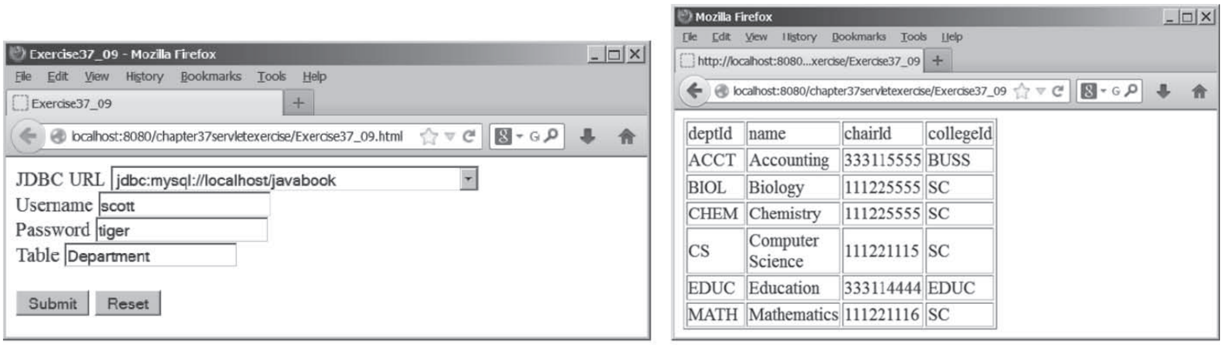
Submit Reset

Hello, John F. Smith, your password has been updated!

(a) (b)

FIGURE 37.30 The user enters the username and the old password and sets a new password. The servlet reports the status of the update to the user.

****37.9** (*Display database tables*) Write an HTML form that prompts the user to enter or select a JDBC driver, database URL, username, password, and table name, as shown in Figure 37.31a. Clicking the *Submit* button displays the table content, as shown in Figure 37.31b.


Exercise37_09 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_09
localhost:8080/chapter37servletexercise/Exercise37_09.html
Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080...xercise/Exercise37_09
localhost:8080/chapter37servletexercise/Exercise37_09

JDBC URL jdbc:mysql://localhost/javabook
Username scott
Password tiger
Table Department

Submit Reset

deptId	name	chairId	collegeId
ACCT	Accounting	333115555	BUSS
BIOL	Biology	111225555	SC
CHEM	Chemistry	111225555	SC
CS	Computer Science	111221115	SC
EDUC	Education	333114444	EDUC
MATH	Mathematics	111221116	SC

(a) (b)

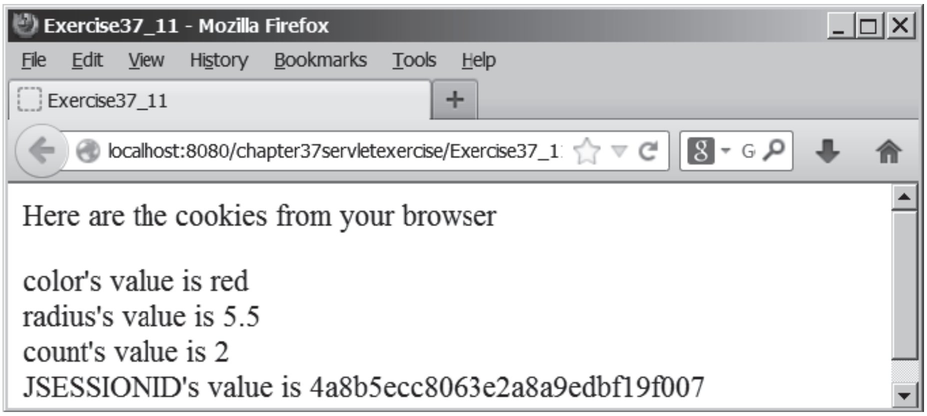
FIGURE 37.31 The user enters database information and specifies a table to display its content.

Section 37.8

***37.10** (*Store cookies*) Write a servlet that stores the following cookies in a browser, and set their max age for two days.

Cookie 1: name is "color" and value is red.
Cookie 2: name is "radius" and value is 5.5.
Cookie 3: name is "count" and value is 2.

***37.11** (*Retrieve cookies*) Write a servlet that displays all the cookies on the client. The client types the URL of the servlet from the browser to display all the cookies stored on the browser. (see Figure 37.32.)


Exercise37_11 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_11
localhost:8080/chapter37servletexercise/Exercise37_1

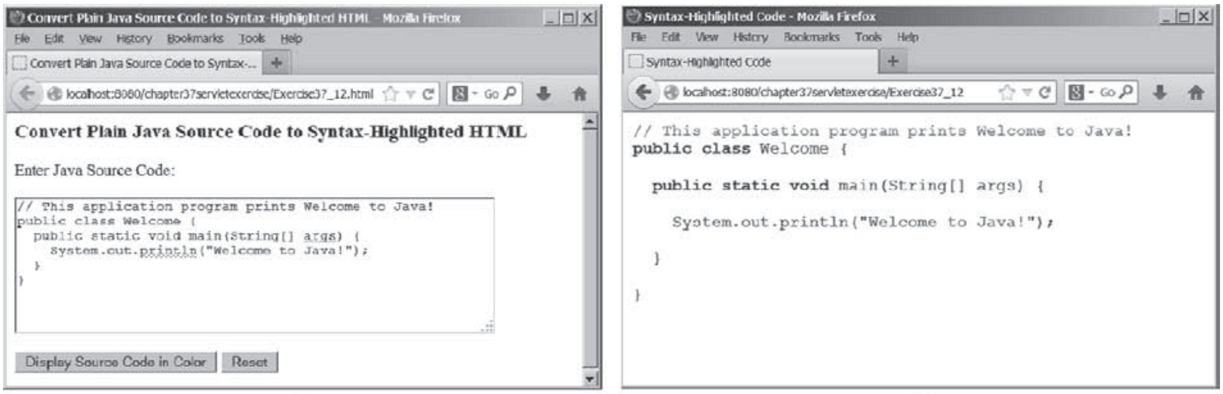
Here are the cookies from your browser

color's value is red
radius's value is 5.5
count's value is 2
JSESSIONID's value is 4a8b5ecc8063e2a8a9edbf19f007

FIGURE 37.32 All the cookies on the client are displayed in the browser.

Comprehensive

*****37.12** (*Syntax highlighting*) Create an HTML form that prompts the user to enter a Java program in a text area, as shown in Figure 37.33a. The form invokes a servlet that displays the Java source code in a syntax-highlighted HTML format, as shown in Figure 37.33b. The keywords, comments, and literals are displayed in bold navy, green, and blue, respectively.


Convert Plain Java Source Code to Syntax-Highlighted HTML - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Convert Plain Java Source Code to Syntax-...
localhost:8080/chapter37servletexercise/Exercise37_12.html
Display Source Code in Color
Reset
Syntax-Highlighted Code - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Syntax-Highlighted Code
localhost:8080/chapter37servletexercise/Exercise37_12

Convert Plain Java Source Code to Syntax-Highlighted HTML

Enter Java Source Code:

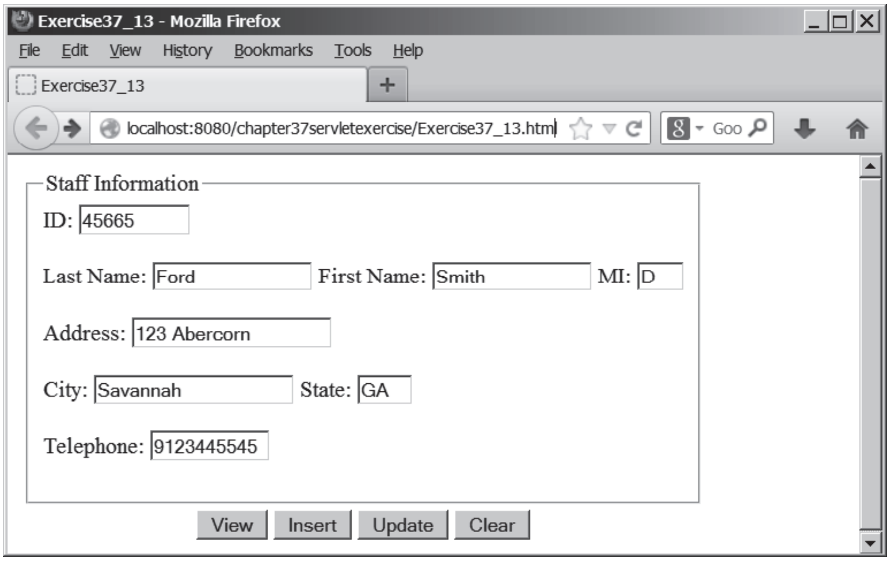
```
// This application program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

```
// This application program prints Welcome to Java!
public class Welcome {

  public static void main(String[] args) {

    System.out.println("Welcome to Java!");

  }

}
```

(a) (b)

FIGURE 37.33 The Java code in plain text in (a) is displayed in HTML with syntax highlighted in (b).

****37.13** (*Access and update a* `Staff` *table*) Write a Java servlet for Exercise 33.1, as shown in Figure 37.34.


Exercise37_13 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_13
localhost:8080/chapter37servletexercise/Exercise37_13.html
View
Insert
Update
Clear

Staff Information

ID: 45665

Last Name: Ford First Name: Smith MI: D

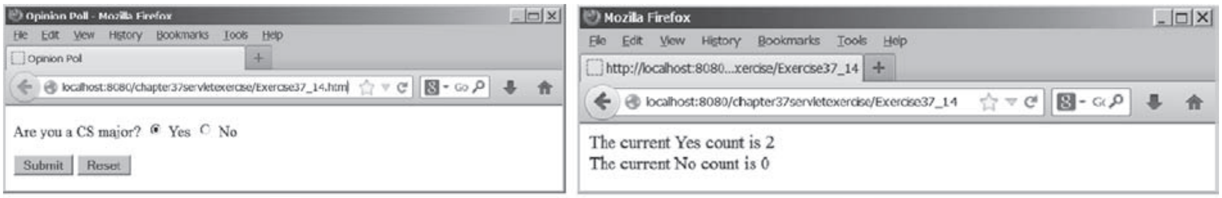
Address: 123 Abercorn

City: Savannah State: GA

Telephone: 9123445545

FIGURE 37.34 The webpage lets you view, insert, and update staff information.

*****37.14** (*Opinion poll*) Create an HTML form that prompts the user to answer a question such as "Are you a CS major?", as shown in Figure 37.35a. When the *Submit* button is clicked, the servlet increases the Yes or No count in a database and displays the current Yes and No counts, as shown in Figure 37.35b.


Opinion Poll - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Opinion Poll
localhost:8080/chapter37servletexercise/Exercise37_14.html
Submit
Reset
Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080...xercise/Exercise37_14
localhost:8080/chapter37servletexercise/Exercise37_14

Are you a CS major? ● Yes ○ No

The current Yes count is 2
The current No count is 0

(a) (b)

FIGURE 37.35 The HTML form prompts the user to enter Yes or No for a question in (a), and the servlet updates the Yes or No counts (b).

Create a table named `Poll`, as follows:

```
create table Poll (
  question varchar(40) primary key,
  yesCount int,
  noCount int);
```

Insert one row into the table, as follows:

```
insert into Poll values ('Are you a CS major? ', 0, 0);
```

CHAPTER

38

JAVASERVER PAGES

Objectives

- To create a simple JSP page (§38.2).
- To explain how a JSP page is processed (§38.3).
- To use JSP constructs to code JSP script (§38.4).
- To use predefined variables and directives in JSP (§§38.5–38.6).
- To use JavaBeans components in JSP (§38.7).
- To get and set JavaBeans properties in JSP (§38.8).
- To associate JavaBeans properties with input parameters (§38.9).
- To forward requests from one JSP page to another (§38.10).
- To develop an application for browsing database tables using JSP (§38.11).

38.1 Introduction

Key Point

JavaServer Pages are the Java scripts and code embedded in an HTML file.

Servlets can be used to generate dynamic Web content. One drawback, however, is that you have to embed HTML tags and text inside the Java source code. Using servlets, you have to modify the Java source code and recompile it if changes are made to the HTML text. If you have a lot of HTML script in a servlet, the program is difficult to read and maintain, since the HTML text is part of the Java program. JavaServer Pages (JSP) was introduced to remedy this drawback. JSP enables you to write regular HTML script in the normal way and embed Java code to produce dynamic content.

38.2 Creating a Simple JSP Page

Key Point

An IDE such an NetBeans is an effecitve tools for creating JavaServer Pages.

JSP provides an easy way to create dynamic webpages and simplify the task of building Web applications. A JavaServer page is like a regular HTML page with special tags, known as *JSP tags*, which enable the Web server to generate dynamic content. You can create a webpage with HTML script and enclose the Java code for generating dynamic content in the JSP tags. Here is an example of a simple JSP page:

```
<!-- CurrentTime.jsp -->
<html>
  <head>
    <title>
      CurrentTime
    </title>
  </head>
  <body>
    Current time is <%= new java.util.Date() %>
  </body>
</html>
```

The dynamic content is enclosed in the tag that begins with `<%=` and ends with `%>`. The current time is returned as a string by invoking the `toString` method of an object of the `java.util.Date` class.

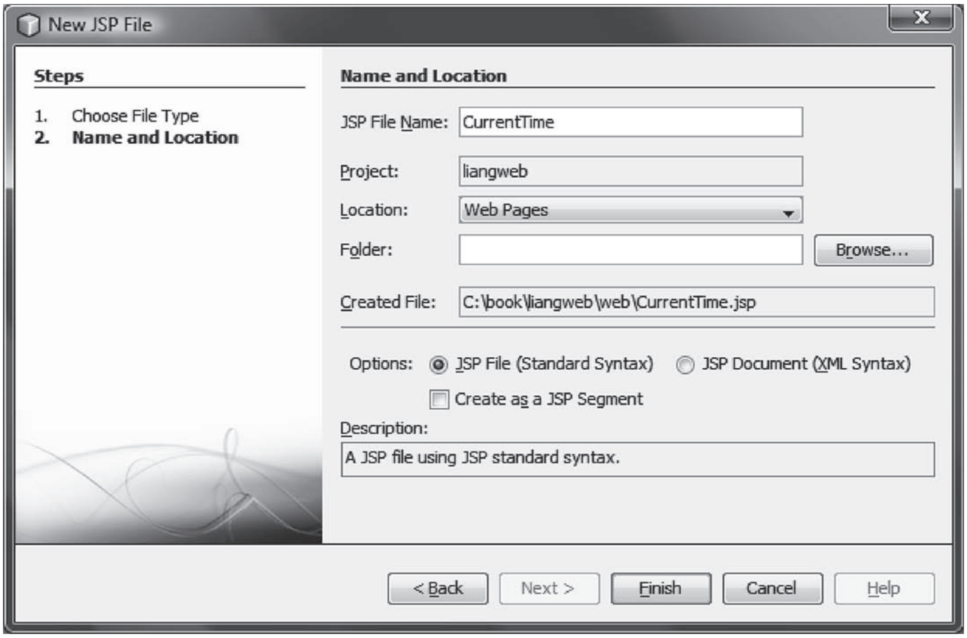
An IDE like NetBeans can greatly simplify the task of developing JSP. To create JSP in NetBeans, first you need to create a Web project. A Web project named `liangweb` was created in the preceding chapter. For convenience, this chapter will create JSP in the `liangweb` project.

Here are the steps to create and run CurrentTime.jsp:

1. Right-click the `liangweb` node in the project pane and choose **New, JSP** to display the New JSP dialog box, as shown in Figure 38.1.
2. Enter `CurrentTime` in the JSP File Name field and click *Finish*. You will see CurrentTime.jsp appearing under the webpages node in `liangweb`.
3. Complete the code for CurrentTime.jsp, as shown in Figure 38.2.
4. Right-click CurrentTime.jsp in the project pane and choose *Run File*. You will see the JSP page displayed in a Web browser, as shown in Figure 38.3.

Note

Like servlets, you can develop JSP in NetBeans, create a .war file, and then deploy the .war file in a Java Web server such as Tomcat and GlassFish.


New JSP File
Steps
Name and Location
< Back
Next >
Finish
Cancel
Help
Browse...

1. Choose File Type
2. **Name and Location**

JSP File Name: CurrentTime

Project: liangweb

Location: Web Pages

Folder:

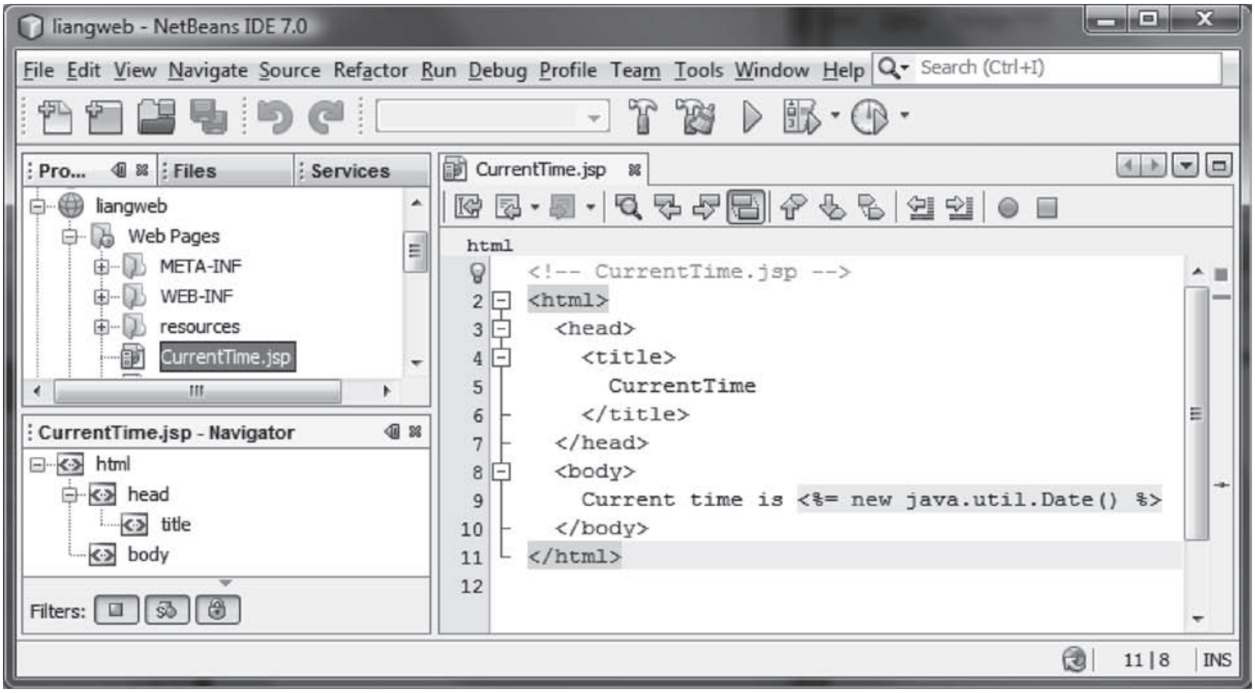
Created File: C:\book\liangweb\web\CurrentTime.jsp

Options: ● JSP File (Standard Syntax) ○ JSP Document (XML Syntax)

☐ Create as a JSP Segment

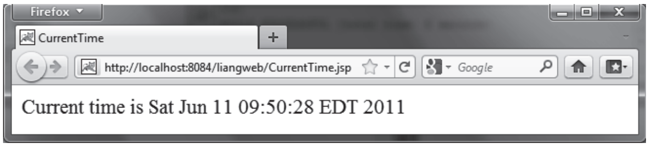
Description: A JSP file using JSP standard syntax.

FIGURE 38.1 You can create a JSP page using NetBeans.


liangweb - NetBeans IDE 7.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
Pro...
Files
Services
liangweb
Web Pages
META-INF
WEB-INF
resources
CurrentTime.jsp
CurrentTime.jsp - Navigator
html
head
title
body
Filters:
CurrentTime.jsp
11 | 8
INS

```
html
<!-- CurrentTime.jsp -->
<html>
  <head>
    <title>
      CurrentTime
    </title>
  </head>
  <body>
    Current time is <%= new java.util.Date() %>
  </body>
</html>
```

FIGURE 38.2 A template for a JSP page is created.


Firefox
CurrentTime
Google

http://localhost:8084/liangweb/CurrentTime.jsp

Current time is Sat Jun 11 09:50:28 EDT 2011

FIGURE 38.3 The result from a JSP page is displayed in a Web browser.

38.3 How Is a JSP Page Processed?

Key Point

JavaServer Pages are preprocessed and compiled into Java servlets by a Java Web server.

A JSP page must first be processed by a Web server before it can be displayed in a Web browser. The Web server must support JSP, and the JSP page must be stored in a file with a .jsp extension. The Web server translates the .jsp file into a Java servlet, compiles the servlet, and executes it. The result of the execution is sent to the browser for display. Figure 38.4 shows how a JSP page is processed by a Web server.

Note

A JSP page is translated into a servlet when the page is requested for the first time. It is not retranslated if the page is not modified. To ensure that the first-time real user does not encounter a delay, JSP developers should test the page after it is installed.

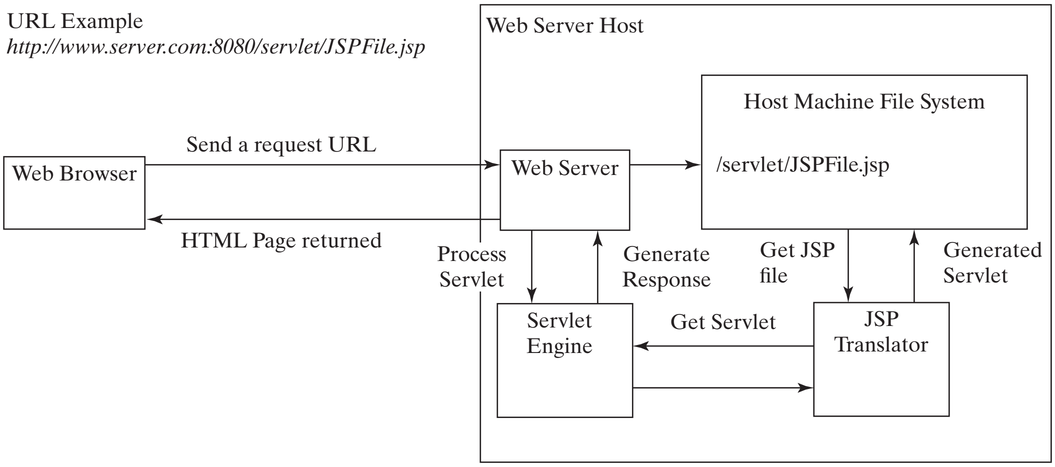

URL Example
http://www.server.com:8080/servlet/JSPFile.jsp
Web Server Host
Host Machine File System
Send a request URL
Web Browser
Web Server
/servlet/JSPFile.jsp
HTML Page returned
Process Servlet
Generate Response
Get JSP file
Generated Servlet
Servlet Engine
Get Servlet
JSP Translator

FIGURE 38.4 A JSP page is translated into a servlet.

Check Point

38.2.1 What is the file-name extension of a JavaServer page? How is a JSP page processed?

38.2.2 Can you create a .war that contains JSP in NetBeans? Where should the .war be placed in a Java application server?

38.2.3 You can display an HTML file (e.g., c:\test.html) by typing the complete file name in the Address field of Internet Explorer. Why can't you display a JSP file by simply typing the file name?

38.4 JSP Scripting Constructs

Key Point

There are three main types of JSP constructs: scripting constructs, directives, and actions.

Scripting elements enable you to specify Java code that will become part of the resultant servlet. *Directives* enable you to control the overall structure of the resultant servlet. *Actions* enable you to control the behavior of the JSP engine. This section introduces scripting constructs.

Three types of JSP scripting constructs can be used to insert Java code into a resultant servlet: expressions, scriptlets, and declarations.

A JSP *expression* is used to insert a Java expression directly into the output. It has the following form:

```
<%= Java expression %>
```

The expression is evaluated, converted into a string, and sent to the output stream of the servlet.

A JSP *scriptlet* enables you to insert a Java statement into the servlet's `jspService` method, which is invoked by the `service` method. A JSP scriptlet has the following form:

```
<% Java statement %>
```

A JSP *declaration* is for declaring methods or fields into the servlet. It has the following form:

```
<%! Java declaration %>
```

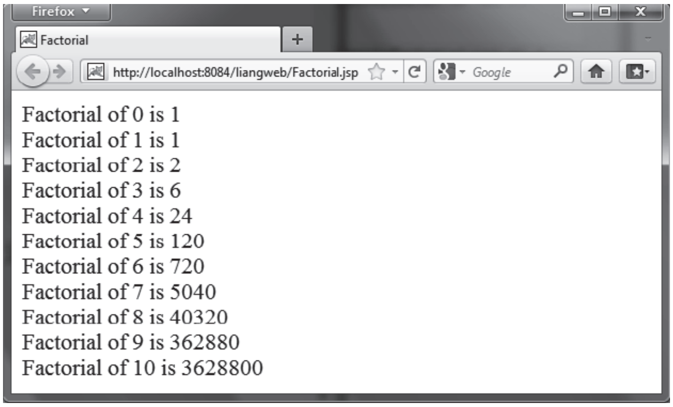
HTML comments have the following form:

```
<!-- HTML Comment -->
```

If you don't want the comment to appear in the resultant HTML file, use the following comment in JSP:

```
<%-- JSP Comment --%>
```

Listing 38.1 creates a JavaServer page that displays factorials for numbers from `0` to `10`, as shown in Figure 38.5.



```
Factorial of 0 is 1
Factorial of 1 is 1
Factorial of 2 is 2
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
Factorial of 6 is 720
Factorial of 7 is 5040
Factorial of 8 is 40320
Factorial of 9 is 362880
Factorial of 10 is 3628800
```

FIGURE 38.5 The JSP page displays factorials.

LISTING 38.1 Factorial.jsp

```
<html>
  <head>
    <title>
      Factorial
    </title>
  </head>
  <body>

  <% for (int i = 0; i <= 10; i++) { %>
      Factorial of <%= i %> is
        <%= computeFactorial(i) %> <br />
  <% } %>

  <%! private long computeFactorial(int n) {
      if (n == 0)
        return 1;
      else
        return n * computeFactorial(n - 1);
19          }
20      %>
21
22      </body>
23    </html>
```

JSP scriptlets are enclosed between `<%` and `%>`. Thus,

```
for (int i = 0; i <= 10; i++) {, (line 9)
```

is a scriptlet and as such is inserted directly into the servlet's `jspService` method.

JSP expressions are enclosed between `<%=` and `%>`. Thus,

```
<%= i %>, (line 10)
```

is an expression and is inserted into the output stream of the servlet.

JSP declarations are enclosed between `<%!` and `%>`. Thus,

```
<%! private long computeFactorial(int n) {
      ...
    }
 %>
```

is a declaration that defines methods or fields in the servlet.

What will be different if line 9 is replaced by the two alternatives shown below? Both work fine, but there is an important difference. In (a), `i` is a local variable in the servlet, whereas in (b), `i` is an instance variable when translated to the servlet.

```
<% int i = 0; %>
<% for ( ; i <= 10; i++) { %>
```

(a)

```
<%! int i; %>
<% for (i = 0; i <= 10; i++) { %>
```

(b)

Caution:

For JSP, the loop body, even though it contains a single statement, must be placed inside braces. It would be wrong to delete the opening brace (`{`) in line 9 and the closing brace (`<% } %>`) in line 12.

Caution:

There is no semicolon at the end of a JSP expression. For example, `<%= i; %>` is incorrect. But there must be a semicolon for each Java statement in a JSP scriptlet. For example, `<% int i = 0 %>` is incorrect.

Caution:

JSP and Java elements are case sensitive, but HTML is not.

Check Point

38.4.1 What are a JSP expression, a JSP scriptlet, and a JSP declaration? How do you write these constructs in JSP?

38.4.2 Find three syntax errors in the following JSP code:

```
<%! int k %>
<% for (int j = 1; j <= 9; j++) %>
   <%= j; %> <br />
```

38.4.3 In the following JSP, which variables are instance variables, and which are local variables when it is translated into in the servlet?

```
<%! int k; %>
<%! int i; %>
```

```
<% for (int j = 1; j <= 9; j++) k += 1;%>
<%= k><br /> <%= i><br /> <%= getTime()><br />
<% private long getTime() {
    long time = System.currentTimeMillis();
    return time;
  } %>
```

38.5 Predefined Variables

JSP provides predefined variables that can be conviniently used in the JSP code.

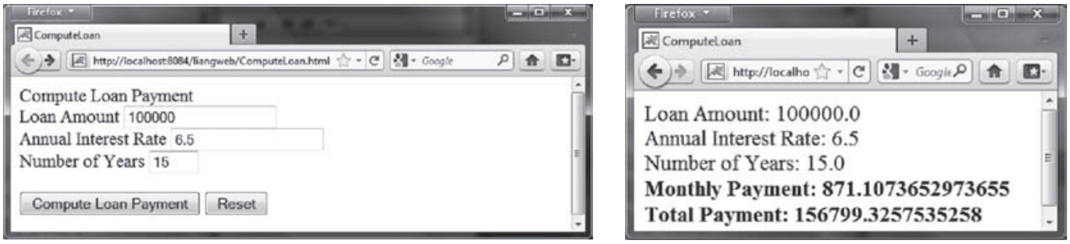
Key Point

You can use variables in JSP. For convenience, JSP provides eight predefined variables from the servlet environment that can be used with JSP expressions and scriptlets. These variables are also known as *JSP implicit objects*.

- `request` represents the client's request, which is an instance of `HttpServletRequest`. You can use it to access request parameters and HTTP headers, such as cookies and host name.
- `response` represents the servlet's response, which is an instance of `HttpServletResponse`. You can use it to set response type and send output to the client.
- `out` represents the character output stream, which is an instance of `PrintWriter` obtained from `response.getWriter()`. You can use it to send character content to the client.
- `session` represents the `HttpSession` object associated with the request, obtained from `request.getSession()`.
- `application` represents the `ServletContext` object for storing persistent data for all clients. The difference between `session` and `application` is that session is tied to one client, but `application` is for all clients to share persistent data.
- `config` represents the `ServletConfig` object for the page.
- `pageContext` represents the `PageContext` object. `PageContext` is a new class introduced in JSP to give a central point of access to many page attributes.
- `page` is an alternative to `this`.

As an example, let us write an HTML page that prompts the user to enter loan amount, annual interest rate, and number of years, as shown in Figure 38.6a. Clicking the *Compute Loan Payment* button invokes a JSP to compute and display the monthly and total loan payments, as shown in Figure 38.6b.

The HTML file is named `ComputeLoan.html` (Listing 38.2). The JSP file is named `ComputeLoan.jsp` (Listing 38.3).



Compute Loan Payment
Loan Amount 100000
Annual Interest Rate 6.5
Number of Years 15

(a)

Loan Amount: 100000.0
Annual Interest Rate: 6.5
Number of Years: 15.0
Monthly Payment: 871.1073652973655
Total Payment: 156799.3257535258

(b)

Figure 38.6 The JSP computes the loan payments.

Listing 38.2 ComputeLoan.html

```
<!-- ComputeLoan.html -->
<html>
  <head>
    <title>ComputeLoan</title>
  </head>
  <body>
    <form method = "get" action = "ComputeLoan.jsp">
    Compute Loan Payment<br />
    Loan Amount
    <input type = "text" name = "loanAmount" /><br />
    Annual Interest Rate
    <input type = "text" name = "annualInterestRate" /><br />
    Number of Years
    <input type = "text" name = "numberOfYears" size = "3" /><br />
    <p><input type = "submit" name = "Submit"
        value = "Compute Loan Payment" />
    <input type = "reset" value = "Reset" /></p>
    </form>
  </body>
</html>
```

Listing 38.3 ComputeLoan.jsp

```
1  <!-- ComputeLoan.jsp -->
2  <html>
3    <head>
4      <title>ComputeLoan</title>
5    </head>
6    <body>
7    <% double loanAmount = Double.parseDouble(
8         request.getParameter("loanAmount"));
9       double annualInterestRate = Double.parseDouble(
10        request.getParameter("annualInterestRate"));
11      double numberOfYears = Integer.parseInt(
12        request.getParameter("numberOfYears"));
13      double monthlyInterestRate = annualInterestRate / 1200;
14      double monthlyPayment = loanAmount * monthlyInterestRate /
15       (1 - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
16      double totalPayment = monthlyPayment * numberOfYears * 12; %>
17   Loan Amount: <%= loanAmount %><br />
18   Annual Interest Rate: <%= annualInterestRate %><br />
19   Number of Years: <%= numberOfYears %><br />
20   <b>Monthly Payment:  <%= monthlyPayment %><br />
21   Total Payment: <%= totalPayment %><br /></b>
22   </body>
23 </html>
```

`ComputeLoan.html` is displayed first to prompt the user to enter the loan amount, annual interest rate, and number of years. Since this file does not contain any JSP elements, it is named with an .html extension as a regular HTML file.

`ComputeLoan.jsp` is invoked upon clicking the *Compute Loan Payment* button in the HTML form. The JSP page obtains the parameter values using the predefined variable `request` in lines 7–12 and computes monthly payment and total payment in lines 13–16. The formula for computing monthly payment is given in Listing 2.9, ComputeLoan.java.

What is wrong if the JSP scriptlet `<%` in line 7 is replaced by the JSP declaration `<%!`? The predefined variables (e.g., `request`, `response`, and `out`) correspond to local variables defined in the servlet methods `doGet` and `doPost`. They must appear in JSP scriptlets, not in JSP declarations.

Tip

`ComputeLoan.jsp` can also be invoked using the following query string: `http://localhost:8084/liangweb/ComputeLoan.jsp?loanAmount=10000&annualInterestRate=6&numberOfYears=15`.

Check Point

38.5.1 Describe the predefined variables in JSP.

38.5.2 What is wrong if the JSP scriptlet `<%` in line 7 in ComputeLoan.jsp (Listing 38.3) is replaced by JSP declaration `<%!`?

38.5.3 Can you use predefined variables (e.g., `request`, `response`, and `out`) in JSP declarations?

38.6 JSP Directives

Key Point

You can use JSP directives to instruct JSP engine on how to process the JSP code.

A JSP directive is a statement that gives the JSP engine information about the JSP page. For example, if your JSP page uses a Java class from a package other than the `java.lang` package, you have to use a directive to import this package. The general syntax for a JSP directive is shown below:

```
<%@ directive attribute = "value" %>, or
<%@ directive attribute1 = "value1"
              attribute2 = "value2"
              ...
              attributen = "valuen" %>
```

The possible directives are:

- `page` lets you provide information for the page, such as importing classes and setting up content type. The `page` directive can appear anywhere in the JSP file.
- `include` lets you insert a file into the servlet when the page is translated to a servlet. The `include` directive must be placed where you want the file to be inserted.
- `taglib` lets you define custom tags.

The following are useful attributes for the `page` directive:

- `import` specifies one or more packages to be imported for this page. For example, the directive `<%@ page import="java.util.*, java.text.*" %>` imports `java.util.*` and `java.text.*`.
- `contentType` specifies the content type for the resultant JSP page. By default, the content type is `text/html` for JSP. The default content type for servlets is `text/plain`.
- `session` specifies a `boolean` value to indicate whether the page is part of the session. By default, `session` is `true`.
- `buffer` specifies the output stream buffer size. By default, it is 8KB. For example, the directive `<%@ page buffer="10KB" %>` specifies that the output buffer size is 10KB. The directive `<%@ page buffer="none" %>` specifies that a buffer is not used.

- `autoFlush` specifies a `boolean` value to indicate whether the output buffer should be automatically flushed when it is full or whether an exception should be raised when the buffer overflows. By default, this attribute is `true`. In this case, the buffer attribute cannot be `none`.
- `isThreadSafe` specifies a `boolean` value to indicate whether the page can be accessed simultaneously without data corruption. By default, it is `true`. If it is set to `false`, the JSP page will be translated to a servlet that implements the `SingleThreadModel` interface.
- `errorPage` specifies a JSP page that is processed when an exception occurs in the current page. For example, the directive `<%@ page errorPage="HandleError.jsp" %>` specifies that HandleError.jsp is processed when an exception occurs.
- `isErrorPage` specifies a `boolean` value to indicate whether the page can be used as an error page. By default, this attribute is `false`.

Listing 38.4 gives an example that shows how to use the page directive to import a class. The example uses the `Loan` class created in Listing 10.2, Loan.java, to simplify Listing 38.3, `ComputeLoan.jsp`. You can create an object of the `Loan` class and use its `monthlyPayment()` and `totalPayment()` methods to compute the monthly payment and total payment.

LISTING 38.4 ComputeLoan1.jsp

```
 1  <!-- ComputeLoan1.jsp -->
 2  <html>
 3    <head>
 4      <title>ComputeLoan Using the Loan Class</title>
 5    </head>
 6  <body>
 7    <%@ page import = "chapter38.Loan" %>
 8     <% double loanAmount = Double.parseDouble(
 9         request.getParameter("loanAmount"));
10       double annualInterestRate = Double.parseDouble(
11         request.getParameter("annualInterestRate"));
12       int numberOfYears = Integer.parseInt(
13         request.getParameter("numberOfYears"));
14       Loan loan =
15         new Loan(annualInterestRate, numberOfYears, loanAmount);
16    %>
17    Loan Amount: <%= loanAmount %><br />
18    Annual Interest Rate: <%= annualInterestRate %><br />
19    Number of Years: <%= numberOfYears %><br />
20    <b>Monthly Payment: <%= loan.getMonthlyPayment() %><br />
21    Total Payment: <%= loan.getTotalPayment() %><br /></b>
22    </body>
23  </html>
```

This JSP uses the `Loan` class. You need to create the class in the liangweb project in package `chapter38` as follows:

```
package chapter38;
public class Loan {
  // Same as lines 2–71 in Listing 10.2, Loan.java, so omitted
```

The directive `<%@ page import ="chapter38.Loan" %>` imports the `Loan` class in line 7. Line 14 creates an object of `Loan` for the given loan amount, annual interest rate, and number of years. Lines 20–21 invoke the `Loan` object's `monthlyPayment()` and `totalPayment()` methods to display monthly payment and total payment.

38.6.1 Describe the JSP directives and attributes for the `page` directive.

38.6.2 If a class does not have a package statement, can you import it?

38.6.3 If you use a custom class from a JSP, where should the class be placed?

38.7 Using JavaBeans in JSP

You can use JavaBeans to create objects for sharing among different JSP pages.

Normally you create an instance of a class in a program and use it in that program. This method is for sharing the class, not the object. JSP allows you to share the object of a class among different pages. To enable an object to be shared, its class must be a JavaBeans component. Recall that this entails the following three features:

- The class is public.
- The class has a public constructor with no arguments.
- The class is serializable. (This requirement is not necessary in JSP.)

To create an instance for a JavaBeans component, use the following syntax:

```
<jsp:useBean id = "objectName" scope = "scopeAttribute"
class = "ClassName" />
```

This syntax is roughly equivalent to

```
<% ClassName objectName = new ClassName() %>
```

except that the `scope` attribute is missing. The scope attribute specifies the scope of the object, and the object is not recreated if it is already within the scope. Listed below are four possible values for the scope attribute:

- `application` specifies that the object is bound to the application. The object can be shared by all sessions of the application.
- `session` specifies that the object is bound to the client's session. Recall that a client's session is automatically created between a Web browser and a Web server. When a client from the same browser accesses two servlets or two JSP pages on the same server, the session is the same.
- `page` is the default scope, which specifies that the object is bound to the page.
- `request` specifies that the object is bound to the client's request.

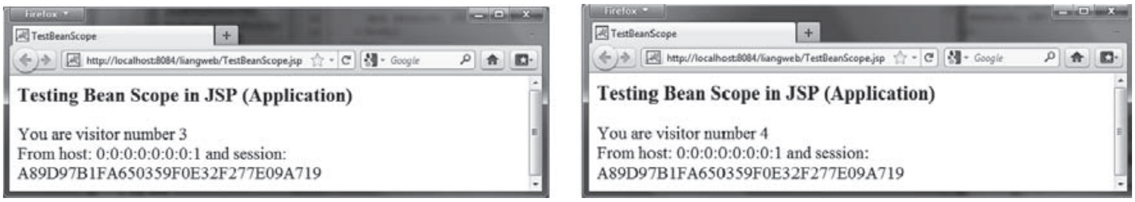
When `<jsp:useBean id="objectName" scope="scopeAttribute" class="ClassName" />` is processed, the JSP engine first searches for an object of the class with the same id and scope. If found, the preexisting bean is used; otherwise, a new bean is created.

Here is another syntax for creating a bean:

```
<jsp:useBean id = "objectName" scope = "scopeAttribute"
  class = "ClassName" >
    statements
</jsp:useBean>
```

The statements are executed when the bean is created. If a bean with the same ID and class name already exists in the scope, the statements are not executed.

Listing 38.5 creates a JavaBeans component named `Count` and uses it to count the number of visits to a JSP page, as shown in Figure 38.7.



Testing Bean Scope in JSP (Application)

You are visitor number 3
From host: 0:0:0:0:0:0:0:1 and session:
A89D97B1FA650359F0E32F277E09A719

Testing Bean Scope in JSP (Application)

You are visitor number 4
From host: 0:0:0:0:0:0:0:1 and session:
A89D97B1FA650359F0E32F277E09A719

FIGURE 38.7 The number of visits to the page is increased when the page is visited.

LISTING 38.5 Count.java

```java
package chapter38;

public class Count {
  private int count = 0;

  /** Return count property */
  public int getCount() {
    return count;
  }

  /** Increase count */
  public void increaseCount() {
   count++;
  }
}
```

The JSP page named TestBeanScope.jsp is created in Listing 38.6.

LISTING 38.6 TestBeanScope.jsp

```
 1  <-- TestBeanScope.jsp -->
 2  <%@ page import = "chapter38.Count" %>
 3  <jsp:useBean id = "count" scope = "application"
 4    class = "chapter38.Count">
 5  </jsp:useBean>
 6  <html>
 7    <head>
 8      <title>TestBeanScope</title>
 9    </head>
10    <body>
11      <h3>Testing Bean Scope in JSP (Application)</h3>
12      <% count.increaseCount(); %>
13      You are visitor number <%= count.getCount() %><br />
14      From host: <%= request.getRemoteHost() %>
15      and session: <%= session.getId() %>
16    </body>
17  </html>
```

The `scope` attribute specifies the scope of the bean. `scope="application"` (line 3) specifies that the bean is alive in the JSP engine and available for all clients to access. The bean can be shared by any client with the directive `<jsp:useBean id="count" scope="application" class="Count">` (lines 3–4). Every client accessing TestBeanScope.jsp causes the count to increase by `1`. The first client causes `count` object to be created, and subsequent access to `TestBeanScope` uses the same object.

If `scope="application"` is changed to `scope="session"`, the scope of the bean is limited to the session from the same browser. The count will increase only if the page is requested from the same browser. If `scope="application"` is changed to `scope="page"`, the scope of the bean is limited to the page, and any other page cannot access this bean. The page will always display count `1`. If `scope="application"` is changed to `scope="request"`, the scope of the bean is limited to the client's request, and any other request on the page will always display count `1`.

If the page is destroyed, the count restarts from `0`. You can fix the problem by storing the count in a random access file or in a database table. Assume you store the count in the `Count` table in a database. The `Count` class can be modified in Listing 38.7.

Listing 38.7 Count.java (Revised Version)

```java
package chapter38;

import java.sql.*;

public class Count {
  private int count = 0;
  private Statement statement = null;

  public Count() {
    initializeJdbc();
  }

  /** Return count property */
  public int getCount() {
    try {
      ResultSet rset = statement.executeQuery
        ("select countValue from Count");
      rset.next();
      count = rset.getInt(1);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }

    return count;
  }

  /** Increase count */
  public void increaseCount() {
    count++;
    try {
      statement.executeUpdate(
        "update Count set countValue = " + count);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  /** Initialize database connection */
  public void initializeJdbc() {
    try {
      Class.forName("com.mysql.jdbc.Driver");

      // Connect to the sample database
      Connection connection = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook" , "scott", "tiger");

      statement = connection.createStatement();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Check Point

38.7.1 You can create an object in a JSP scriptlet. What is the difference between an object created using the `new` operator and a bean created using the `<jsp:useBean ... >` tag?

38.7.2 What is the `scope` attribute for? Describe four scope attributes.

38.7.3 Describe how a `<jsp:useBean ... >` statement is processed by the JSP engine.

38.8 Getting and Setting Properties

Key Point

JSP provides convenient syntax for getting and setting JavaBeans properties.

By convention, a JavaBeans component provides the get and set methods for reading and modifying its private properties. You can get the property in JSP using the syntax shown below:

```
<jsp:getProperty name = "beanId" property = "sample" />
```

This is roughly equivalent to

```
<%= beanId.getSample() %>
```

You can set the property in JSP using the following syntax:

```
<jsp:setProperty name = "beanId"
  property = "sample" value = "test1" />
```

This is equivalent to

```
<% beanId.setSample("test1"); %>
```

38.9 Associating Properties with Input Parameters

Often properties are associated with input parameters. Suppose you want to get the value of the input parameter named `score` and set it to the JavaBeans property named `score`. You could write the following code:

```
<% double score = Double.parseDouble(
     request.getParameter("score")); %>
<jsp:setProperty name = "beanId"  property = "score"
     value = "<%= score %>" />
```

This is cumbersome. JSP provides a convenient syntax that can be used to simplify it:

```
<jsp:setProperty name = "beanId" property = "score"
   param = "score" />
```

Instead of using the `value` attribute, you use the `param` attribute to name an input parameter. The value of this parameter is set to the property.

Note

Simple type conversion is performed automatically when a bean property is associated with an input parameter. A string input parameter is converted to an appropriate primitive data type or a wrapper class for a primitive type. For example, if the bean property is of the `int` type, the value of the parameter will be converted to the `int` type. If the bean property is of the `Integer` type, the value of the parameter will be converted to the `Integer` type.

Often the bean property and the parameter have the same name. You can use the following convenient statement to associate all the bean properties in `beanId` with the parameters that match the property names:

```
<jsp:setProperty name = "beanId" property = "*" />
```

38.9.1 Example: Computing Loan Payments Using JavaBeans

This example uses JavaBeans to simplify Listing 38.4, ComputeLoan1.jsp, by associating the bean properties with the input parameters. The new ComputeLoan2.jsp is given in Listing 38.8.

LISTING 38.8 `ComputeLoan2.jsp`

```
 1  <!-- ComputeLoan2.jsp -->
 2  <html>
 3    <head>
 4      <title>ComputeLoan Using the Loan Class</title>
 5    </head>
 6    <body>
 7      <%@ page import = "chapter38.Loan" %>
 8      <jsp:useBean id = "loan" class = "chapter38.Loan"
 9        scope = "page" ></jsp:useBean>
10      <jsp:setProperty name = "loan" property = "*" />
11      Loan Amount: <%= loan.getLoanAmount() %><br />
12      Annual Interest Rate: <%= loan.getAnnualInterestRate() %><br />
13      Number of Years: <%= loan.getNumberOfYears() %><br />
14      <b>Monthly Payment: <%= loan.monthlyPayment() %><br />
15      Total Payment: <%= loan.totalPayment() %><br /></b>
16    </body>
17  </html>
```

Lines 8–9

```
<jsp:useBean id = "loan" class = "chapter38.Loan"
  scope = "page" ></jsp:useBean>
```

create a bean named `loan` for the `Loan` class. Line 10

```
<jsp:setProperty name = "loan" property = "*" />
```

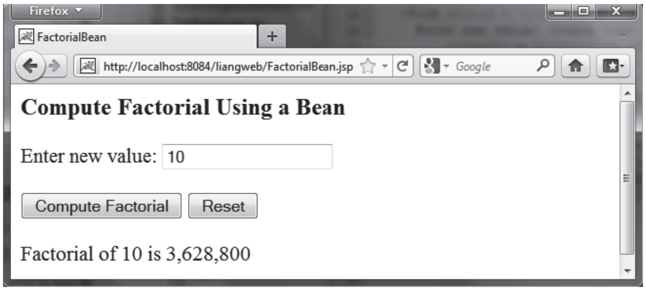
associates the bean properties `loanAmount`, `annualInteresteRate`, and `numberOfYears` with the input parameter values and performs type conversion automatically.

Lines 11–13 use the accessor methods of the loan bean to get the loan amount, annual interest rate, and number of years.

This program acts the same as in Listings 38.3 and 38.4, `ComputeLoan.jsp` and `ComputeLoan1.jsp`, but the coding is much, more simplified.

38.9.2 Example: Computing Factorials Using JavaBeans

This example creates a JavaBeans component named `FactorialBean` and uses it to compute the factorial of an input number in a JSP page named FactorialBean.jsp, as shown in Figure 38.8.


Firefox
FactorialBean
http://localhost:8084/liangweb/FactorialBean.jsp
Google
Compute Factorial
Reset

Compute Factorial Using a Bean

Enter new value: 10

Factorial of 10 is 3,628,800

Figure 38.8 The factorial of an input integer is computed using a method in `FactorialBean`.

Create a JavaBeans component named `FactorialBean.java` (Listing 38.9). Create FactorialBean.jsp (Listing 38.10).

Listing 38.9 FactorialBean.java

```java
package chapter38;

public class FactorialBean {
  private int number;

  /** Return number property */
  public int getNumber() {
    return number;
  }

  /** Set number property */
  public void setNumber(int newValue) {
    number = newValue;
  }

  /** Obtain factorial */
  public long getFactorial() {
    long factorial = 1;
    for (int i = 1; i <= number; i++)
      factorial *= i;
    return factorial;
  }
}
```

Listing 38.10 FactorialBean.jsp

```jsp
<!-- FactorialBean.jsp -->
<%@ page import = "chapter38.FactorialBean" %>
<jsp:useBean id = "factorialBeanId"
  class = "chapter38.FactorialBean" scope = "page" >
</jsp:useBean>
<jsp:setProperty name = "factorialBeanId" property = "*" />
<html>
  <head>
    <title>
      FactorialBean
    </title>
  </head>
  <body>
  <h3>Compute Factorial Using a Bean</h3>
```

```
15      <form method = "post">
16        Enter new value: <input name = "number" /><br /><br />
17        <input type = "submit" name = "Submit"
18          value = "Compute Factorial" />
19        <input type = "reset" value = "Reset" /><br /><br />
20        Factorial of
21          <jsp:getProperty name = "factorialBeanId"
22            property = "number" /> is
23          <%@ page import = "java.text.*" %>
24          <% NumberFormat format = NumberFormat.getNumberInstance(); %>
25          <%= format.format(factorialBeanId.getFactorial()) %>
26       </form>
27     </body>
28   </html>
```

The `jsp:useBean` tag (lines 3–4) creates a bean `factorialBeanId` of the `FactorialBean` class. Line 5 `<jsp:setProperty name="factorialBeanId" property="*" />` associates all the bean properties with the input parameters that have the same name. In this case, the bean property `number` is associated with the input parameter `number`. When you click the *Compute Factorial* button, JSP automatically converts the input value for `number` from string into `int` and sets it to `factorialBean` before other statements are executed.

Lines 21–22 `<jsp:getProperty name="factorialBeanId" property="number" />` tag (line 21) is equivalent to `<%= factorialBeanId.getNumber() %>`. The method `factorialBeanId.getFactorial()` (line 25) returns the factorial for the number in `factorialBeanId`.

Design Guide

Mixing a lot of Java code with HTML in a JSP page makes the code difficult to read and to maintain. You should move the Java code to a .java file as much as you can.

Following the preceding design guide, you may improve the preceding example by moving the Java code in lines 23–25 to the `FactorialBean` class. The new `FactorialBean.java` and `FactorialBean.jsp` are given in Listings 38.11 and 38.12.

LISTING 38.11 NewFactorialBean.java

```
package chapter38;

import java.text.*;

public class NewFactorialBean {
  private int number;

  /** Return number property */
  public int getNumber() {
    return number;
  }

  /** Set number property */
  public void setNumber(int newValue) {
    number = newValue;
  }

  /** Obtain factorial */
  public long getFactorial() {
    long factorial = 1;
    for (int i = 1; i <= number; i++)
      factorial *= i;
    return factorial;
  }

  /** Format number */
  public static String format(long number) {
    NumberFormat format = NumberFormat.getNumberInstance();
    return format.format(number);
  }
}
```

Listing 38.12 NewFactorialBean.jsp

```
<!-- NewFactorialBean.jsp -->
<%@ page import = "chapter38.NewFactorialBean" %>
<jsp:useBean id = "factorialBeanId"
  class = "chapter38.NewFactorialBean" scope = "page" >
</jsp:useBean>
<jsp:setProperty name = "factorialBeanId" property = "*" />
<html>
  <head>
    <title>
      FactorialBean
    </title>
  </head>
  <body>
  <h3>Compute Factorial Using a Bean</h3>
  <form method = "post">
    Enter new value: <input name = "number" /><br /><br />
    <input type = "submit" name = "Submit"
      value = "Compute Factorial" />
    <input type = "reset" value = "Reset" /><br /><br />
      Factorial of
        <jsp:getProperty name = "factorialBeanId"
          property = "number" /> is
        <%= NewFactorialBean.format(factorialBeanId.getFactorial()) %>
    </form>
  </body>
</html>
```

There is a problem in this page. The program cannot display large factorials. For example, if you entered value `21`, the program would display an incorrect factorial. To fix this problem, all you need to do is to revise the `NewFactorialBean` class using `BigInteger` to computing factorials (see Exercise 38.18).

38.9.3 Example: Displaying International Time

Listing 37.5, TimeForm.java, gives a Java servlet that uses the `doGet` method to generate an HTML form for the user to specify a locale and time zone (see Figure 37.18a) and uses the `doPost` method to display the current time for the specified time zone in the specified locale (see Figure 37.18b). This section rewrites the servlet using JSP. You have to create two JSP pages, one for displaying the form, and the other for displaying the current time.

In the `TimeForm.java` servlet, arrays `allLocale` and `allTimeZone` are the data fields. The `doGet` and `doPost` methods both use the arrays. Since the available locales and time zones are used in both pages, it is better to create an object that contains all available locales and time zones. This object can be shared by both pages.

Let us create a JavaBeans component named TimeBean.java (Listing 38.13). This class obtains all the available locales in an array in line 7 and all time zones in an array in line 8. The bean properties `localeIndex` and `timeZoneIndex` (lines 9–10) are defined to refer to an element in the arrays. The `currentTimeString()` method (lines 42–52) returns a string for the current time with the specified locale and time zone.

LISTING 38.13 TimeBean.java

```
 1  package chapter38;
 2
 3  import java.util.*;
 4  import java.text.*;
 5
 6  public class TimeBean {
 7    private Locale[] allLocale = Locale.getAvailableLocales();
 8    private String[] allTimeZone = TimeZone.getAvailableIDs();
 9    private int localeIndex;
10    private int timeZoneIndex;
11
12    public TimeBean() {
13      Arrays.sort(allTimeZone);
14    }
15
16    public Locale[] getAllLocale() {
17      return allLocale;
18    }
19
20    public String[] getAllTimeZone() {
21      return allTimeZone;
22    }
23
24    public int getLocaleIndex() {
25      return localeIndex;
26    }
27
28    public int getTimeZoneIndex() {
29      return timeZoneIndex;
30    }
31
32    public void setLocaleIndex(int index) {
33      localeIndex = index;
34    }
35
36    public void setTimeZoneIndex(int index) {
37      timeZoneIndex = index;
38    }
39
40    /** Return a string for the current time
41     * with the specified locale and time zone */
42    public String currentTimeString(
43        int localeIndex, int timeZoneIndex) {
44      Calendar calendar =
45        new GregorianCalendar(allLocale[localeIndex]);
46      TimeZone timeZone =
47        TimeZone.getTimeZone(allTimeZone[timeZoneIndex]);
48      DateFormat dateFormat = DateFormat.getDateTimeInstance(
49        DateFormat.FULL, DateFormat.FULL, allLocale[localeIndex]);
50      dateFormat.setTimeZone(timeZone);
51      return dateFormat.format(calendar.getTime());
52    }
53  }
```

Create DisplayTimeForm.jsp (Listing 38.14). This page displays a form just like the one shown in Figure 37.18a. Line 2 imports the `TimeBean` class. A bean is created in lines 3–5 and is used in lines 17, 19, 24, and 26 to return all locales and time zones. The scope of the bean is application (line 4), so the bean can be shared by all sessions of the application.

LISTING 38.14 DisplayTimeForm.jsp

```
1  <!-- DisplayTimeForm.jsp -->
2  <%@ pageimport = "chapter38.TimeBean" %>
3  <jsp:useBean id = "timeBeanId"
4    class = "chapter38.TimeBean" scope = "application" >
5  </jsp:useBean>
6
7  <html>
8    <head>
9      <title>
10       Display Time Form
11     </title>
12   </head>
13   <body>
14   <h3>Choose locale and time zone</h3>
15   <form method = "post" action = "DisplayTime.jsp">
16     Locale <select size = "1" name = "localeIndex">
17     <% for (int i = 0; i < timeBeanId.getAllLocale().length; i++) {%>
18            <option value = "<%= i %>">
19              <%= timeBeanId.getAllLocale()[i] %>
20            </option>
21     <%}%>
22            </select><br />
23     Time Zone <select size = "1" name = "timeZoneIndex">
24     <% for (int i = 0; i < timeBeanId.getAllTimeZone().length; i++) {%>
25            <option value = "<%= i %>">
26              <%= timeBeanId.getAllTimeZone()[i] %>
27            </option>
28         <%}%>
29         </select><br />
30     <input type = "submit" name = "Submit"
31       value = "Get Time" />
32     <input type = "reset" value = "Reset" />
33     </form>
34   </body>
35 </html>
```

Create DisplayTime.jsp (Listing 38.15). This page is invoked from `DisplayTimeForm.jsp` to display the time with the specified locale and time zone, just as in Figure 37.18b.

LISTING 38.15 DisplayTime.jsp

```
<!-- DisplayTime.jsp -->
<%@page pageEncoding = "GB18030"%>
<%@ page import = "chapter38.TimeBean" %>
<jsp:useBean id = "timeBeanId"
  class = "chapter38.TimeBean" scope = "application" >
</jsp:useBean>
<jsp:setProperty name = "timeBeanId" property = "*" />

<html>
  <head>
```

```
11        <title>
12          Display Time
13        </title>
14      </head>
15      <body>
16      <h3>Choose locale and time zone</h3>
17        Current time is
18          <%= timeBeanId.currentTimeString(timeBeanId.getLocaleIndex(),
19            timeBeanId.getTimeZoneIndex()) %>
20      </body>
21    <html>
```

Line 2 sets the character encoding for the page to GB18030 for displaying international characters. By default, it is UTF-8.

Line 5 imports `chapter38.TimeBean` and creates a bean using the same id as in the preceding page. Since the object is already created in the preceding page, the `timeBeanId` in this page (lines 4–6) and in the preceding page point to the same object.

38.9.4 Example: Registering Students

Listing 37.11, RegistrationWithHttpSession.java, gives a Java servlet that obtains student information from an HTML form (see Figure 37.21) and displays the information for user confirmation (see Figure 37.22). Once the user confirms it, the servlet stores the data into the database. This section rewrites the servlet using JSP. You will create two JSP pages, one named GetRegistrationData.jsp for displaying the data for user confirmation and the other named StoreData.jsp for storing the data into the database.

Since every session needs to connect to the same database, you should declare a class for connecting to the database and for storing a student to the database. This class named `StoreData` is given in Listing 38.16. The `initializeJdbc` method (lines 15–31) connects to the database and creates a prepared statement for storing a record to the Address table. The `storeStudent` method (lines 34–45) executes the prepared statement to store a student address. The `Address` class is created in Listing 37.12.

LISTING 38.16 StoreData.java

```
 1  package chapter38;
 2
 3  import java.sql.*;
 4  import chapter37.Address;
 5
 6  public class StoreData {
 7    // Use a prepared statement to store a student into the database
 8    private PreparedStatement pstmt;
 9
10    public StoreData() {
11      initializeJdbc();
12    }
13
14    /** Initialize database connection */
15    private void initializeJdbc() {
16      try {
17        Class.forName("com.mysql.jdbc.Driver");
18
19        // Connect to the sample database
20        Connection connection = DriverManager.getConnection
21          ("jdbc:mysql://localhost/javabook" , "scott", "tiger");
22
```

```
      // Create a Statement
      pstmt = connection.prepareStatement("insert into Address " +
        "(lastName, firstName, mi, telephone, email, street, city, "
        + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }

  /** Store a student record to the database */
  public void storeStudent(Address address) throws SQLException {
    pstmt.setString(1, address.getLastName());
    pstmt.setString(2, address.getFirstName());
    pstmt.setString(3, address.getMi());
    pstmt.setString(4, address.getTelephone());
    pstmt.setString(5, address.getEmail());
    pstmt.setString(6, address.getStreet());
    pstmt.setString(7, address.getCity());
    pstmt.setString(8, address.getState());
    pstmt.setString(9, address.getZip());
    pstmt.executeUpdate();
  }
}
```

The HTML file that displays the form is identical to `Registration.html` in Listing 37.8 except that the action is replaced by HGetRegistrationData.jsp.

`GetRegistrationData.jsp`, which obtains the data from the form, is shown in Listing 38.17. A bean is created in lines 3–4. Line 5 obtains the property values from the form. This is a shorthand notation. Note the parameter names and the property names must be the same to use this notation.

LISTING 38.17 `GetRegistrationData.jsp`

```
 1  <!-- GetRegistrationData.jsp -->
 2  <%@ page import = "chapter37.Address" %>
 3  <jsp:useBean id = "addressId"
 4    class = "chapter37.Address" scope = "session"></jsp:useBean>
 5  <jsp:setProperty name = "addressId" property = "*" />
 6
 7  <html>
 8    <body>
 9      <h1>Registration Using JSP</h1>
10
11      <%
12      if (addressId.getLastName() == null ||
13          addressId.getFirstName() == null) {
14        out.println("Last Name and First Name are required");
15        return; // End the method
16      }
17      %>
18
19      <p>You entered the following data</p>
20      <p>Last name: <%= addressId.getLastName() %></p>
21      <p>First name: <%= addressId.getFirstName() %></p>
22      <p>MI: <%= addressId.getMi() %></p>
23      <p>Telephone: <%= addressId.getTelephone() %></p>
```

```
24       <p>Email: <%= addressId.getEmail() %></p>
25       <p>Address: <%= addressId.getStreet() %></p>
26       <p>City: <%= addressId.getCity() %></p>
27       <p>State: <%= addressId.getState() %></p>
28       <p>Zip:  <%= addressId.getZip() %></p>
29
30       <!-- Set the action for processing the answers -->
31       <form method = "post" action = "StoreStudent.jsp">
32         <input type = "submit" value = "Confirm">
33       </form>
34     </body>
35   </html>
```

`GetRegistrationData.jsp` invokes `StoreStudent.jsp` (line 31) when the user clicks the *Confirm* button. In Listing 38.18, the same `addressId` is shared with the preceding page within the scope of the same session in lines 3–4. A bean for `StoreData` is created in lines 5–6 with the scope of application.

Listing 38.18 StoreStudent.jsp

```
 1   <!-- StoreStudent.jsp -->
 2   <%@ page import = "chapter37.Address" %>
 3   <jsp:useBean id = "addressId" class = "chapter37.Address"
 4     scope = "session"></jsp:useBean>
 5   <jsp:useBean id = "storeDataId" class = "chapter38.StoreData"
 6     scope = "application"></jsp:useBean>
 7
 8   <html>
 9     <body>
10       <%
11         storeDataId.storeStudent(addressId);
12
13         out.println(addressId.getFirstName() + " " +
14           addressId.getLastName() +
15           " is now registered in the database");
16         out.close(); // Close stream
17       %>
18     </body>
19   </html>
```

Note

The scope for `addressId` is *session*, but the scope for `storeDataId` is *application*. Why? GetRegistrationData.jsp obtains student information, and StoreData.jsp stores the information in the same session. So the *session* scope is appropriate for `addressId`. All the sessions access the same database and use the same prepared statement to store data. With the *application* scope for `storeDataId`, the bean for `StoreData` needs to be created just once.

Note

The `storeStudent` method in line 11 may throw a `java.sql.SQLException`. In JSP, you can omit the try-block for checked exceptions. In case of an exception, JSP displays an error page.

Tip

Using beans is an effective way to develop JSP. You should put Java code into a bean as much as you can. The bean not only simplifies JSP programming but also makes code reusable. The bean can also be used to implement persistent sessions.

38.10 Forwarding Requests from JavaServer Pages

Key Point

You can use the JSP forward tag to jump to navigate to another HTML page.

Web applications developed using JSP generally consist of many pages linked together. JSP provides a forwarding tag in the following syntax that can be used to forward a page to another page:

```
<jsp:forward page = "destination" />
```

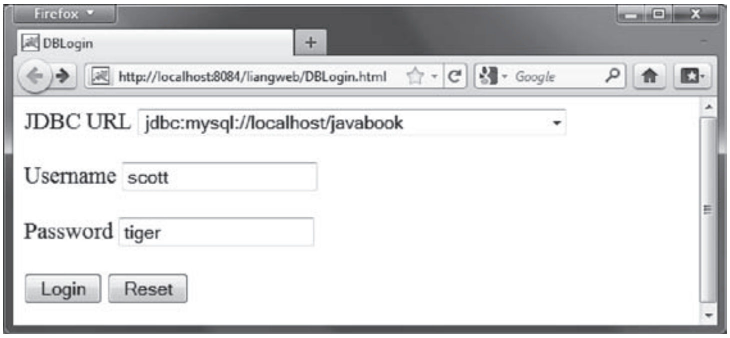
Check Point

38.10.1 How do you associate bean properties with input parameters?

38.10.2 How do you write a statement to forward requests to another JSP page?

38.11 Case Study: Browsing Database Tables

This section presents a very useful JSP application for browsing tables. When you start the application, the first page prompts the user to enter the JDBC driver, URL, username, and password for a database, as shown in Figure 38.9. After you log in to the database, you can select a table to browse, as shown in Figure 38.10. Clicking the *Browse Table Content* button displays the table content, as shown in Figure 38.11.

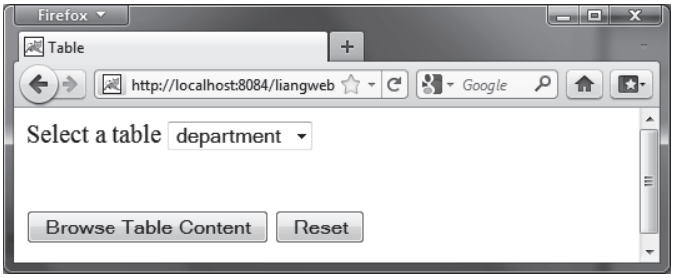

Firefox
DBLogin
http://localhost:8084/liangweb/DBLogin.html
Google
Login
Reset

JDBC URL jdbc:mysql://localhost/javabook

Username scott

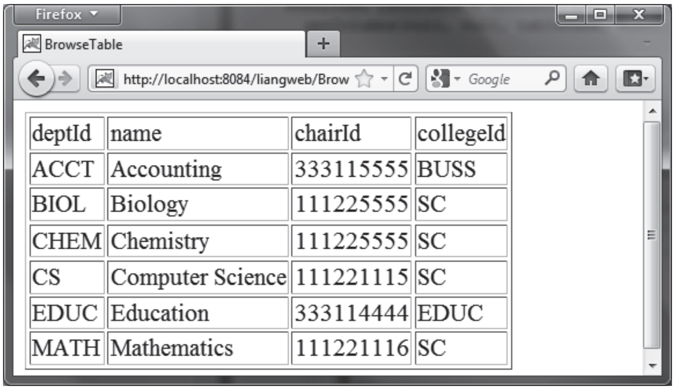
Password tiger

FIGURE 38.9 To access a database, you need to provide the JDBC driver, URL, username, and password.


Firefox
Table
http://localhost:8084/liangweb
Google
Browse Table Content
Reset

Select a table department

FIGURE 38.10 You can select a table to browse from this page.



deptId	name	chairId	collegeId
ACCT	Accounting	333115555	BUSS
BIOL	Biology	111225555	SC
CHEM	Chemistry	111225555	SC
CS	Computer Science	111221115	SC
EDUC	Education	333114444	EDUC
MATH	Mathematics	111221116	SC

FIGURE 38.11 The contents of the selected table are displayed.

Create a JavaBeans component named `DBBean.java` (see Listing 38.19).

LISTING 38.19 DBBean.java

```java
package chapter38;

import java.sql.*;

public class DBBean {
  private Connection connection = null;
  private String username;
  private String password;
  private String driver;
  private String url;

  /** Initialize database connection */
  public void initializeJdbc() {
    try {
      System.out.println("Driver is " + driver);
      Class.forName(driver);

      // Connect to the sample database
      connection = DriverManager.getConnection(url, username,
        password);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  /** Get tables in the database */
  public String[] getTables() {
    String[] tables = null;

    try {
      DatabaseMetaData dbMetaData = connection.getMetaData();
      ResultSet rsTables = dbMetaData.getTables(null, null, null,
        new String[] {"TABLE"});

      int size = 0;
      while (rsTables.next()) size++;

      rsTables = dbMetaData.getTables(null, null, null,
        new String[] {"TABLE"});

      tables = new String[size];
      int i = 0;
      while (rsTables.next())
        tables[i++] = rsTables.getString("TABLE_NAME");
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }

    return tables;
  }

  /** Return connection property */
  public Connection getConnection() {
    return connection;
  }

  public void setUsername(String newUsername) {
    username = newUsername;
  }

  public String getUsername() {
    return username;
  }

  public void setPassword(String newPassword) {
    password = newPassword;
  }

  public String getPassword() {
    return password;
  }

  public void setDriver(String newDriver) {
    driver = newDriver;
  }

  public String getDriver() {
    return driver;
  }

  public void setUrl(String newUrl) {
    url = newUrl;
  }

  public String getUrl() {
    return url;
  }
}
```

Create an HTML file named `DBLogin.html` (see Listing 38.20) that prompts the user to enter database information and three JSP files named `DBLoginInitialization.jsp` (see Listing 38.21), `Table.jsp` (see Listing 38.22), and `BrowseTable.jsp` (see Listing 38.23) to process and obtain database information.

LISTING 38.20 DBLogin.html

```
<!-- DBLogin.html -->
<html>
  <head>
    <title>
      DBLogin
    </title>
  </head>
  <body>
    <form method = "post" action = "DBLoginInitialization.jsp">
    JDBC URL
    <select name = "url" size = "1">
      <option>jdbc:odbc:ExampleMDBDataSource</option>
      <option>jdbc:mysql://localhost/javabook</option>
      <option>jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl</option>
    </select><br /><br />
    Username <input name = "username" /><br /><br />
    Password <input name = "password" /><br /><br />
    <input type = "submit" name = "Submit" value = "Login" />
    <input type = "reset" value = "Reset" />
    </form>
  </body>
</html>
```

LISTING 38.21 DBLoginInitialization.jsp

```
<!-- DBLoginInitialization.jsp -->
<%@ page import = "chapter38.DBBean" %>
<jsp:useBean id = "dBBeanId" scope = "session"
  class = "chapter38.DBBean">
</jsp:useBean>
<jsp:setProperty name = "dBBeanId" property = "*" />
<html>
  <head>
    <title>DBLoginInitialization</title>
  </head>
  <body>

  <%-- Connect to the database --%>
  <% dBBeanId.initializeJdbc(); %>

  <% if (dBBeanId.getConnection() == null) { %>
    Error: Login failed. Try again.
  <% }
    else {%>
      <jsp:forward page = "Table.jsp" />
  <% } %>
  </body>
</html>
```

LISTING 38.22 Table.jsp

```
<!-- Table.jsp -->
<%@ page import = "chapter38.DBBean" %>
<jsp:useBean id = "dBBeanId" scope = "session"
  class = "chapter38.DBBean">
</jsp:useBean>
<html>
  <head>
```

```
    <title>Table</title>
  </head>
  <body>
  <% String[] tables = dBBeanId.getTables();
    if (tables == null) { %>
      No tables
  <% }
    else { %>
      <form method = "post" action = "BrowseTable.jsp">
        Select a table
         <select name = "tablename" size = "1">
      <% for (int i = 0; i < tables.length; i++) { %>
            <option><%= tables[i] %></option>
      <% }
    } %>
          </select><br /><br /><br />
    <input type = "submit" name = "Submit"
      value = "Browse Table Content">
    <input type = "reset" value = "Reset">
    </form>
  </body>
</html>
```

LISTING 38.23 BrowseTable.jsp

```
<!-- BrowseTable.jsp -->
<%@ page import = "chapter38.DBBean" %>
<jsp:useBean id = "dBBeanId" scope = "session"
  class = "chapter38.DBBean" >
</jsp:useBean>
<%@ page import = "java.sql.*" %>
<html>
  <head>
    <title>BrowseTable</title>
  </head>
  <body>

  <% String tableName = request.getParameter("tablename");

      ResultSet rsColumns = dBBeanId.getConnection().getMetaData().
      getColumns(null, null, tableName, null);
  %>
  <table border = "1">
    <tr>
      <% // Add column names to the table
      while (rsColumns.next()) { %>
        <td><%= rsColumns.getString("COLUMN_NAME") %></td>
    <%}%>
    </tr>

    <% Statement statement =
         dBBeanId.getConnection().createStatement();
       ResultSet rs = statement.executeQuery(
         "select * from " + tableName);

      // Get column count
      int columnCount = rs.getMetaData().getColumnCount();

      // Store rows to rowData
      while (rs.next()) {
```

```
    out.println("<tr>");
    for (int i = 0; i < columnCount; i++) { %>
     <td><%= rs.getObject(i + 1) %></td>
  <%  }
    out.println("</tr>");
   } %>
 </table>
 </body>
</html>
```

You start the application from DBLogin.html. This page prompts the user to enter a JDBC driver, URL, username, and password to log in to a database. A list of accessible drivers and URLs is provided in the selection list. You must make sure that these database drivers are added into the Libraries node in the project.

When you click the *Login* button, DBLoginInitialization.jsp is invoked. When this page is processed for the first time, an instance of `DBBean` named `dBBeanId` is created. The input parameters `driver`, `url`, `username`, and `password` are passed to the bean properties. The `initializeJdbc` method loads the driver and establishes a connection to the database. If login fails, the `connection` property is `null`. In this case, an error message is displayed. If login succeeds, control is forwarded to Table.jsp.

Table.jsp shares `dBBeanId` with DBLoginInitialization.jsp in the same session, so it can access `connection` through `dBBeanId` and obtain tables in the database using the database metadata. The table names are displayed in a selection box in a form. When the user selects a table name and clicks the *Browse Table Content* button, BrowseTable.jsp is processed.

BrowseTable.jsp shares `dBBeanId` with Table.jsp and DBLoginInitialization.jsp in the same session. It retrieves the table contents for the selected table from Table.jsp.

JSP Scripting Constructs Syntax

- `<%= Java expression %>` The expression is evaluated and inserted into the page.
- `<% Java statement %>` Java statements inserted in the `jspService` method.
- `<%! Java declaration %>` Defines data fields and methods.
- `<%-- JSP comment %>` The JSP comments do not appear in the resultant HTML file.
- `<%@ directive attribute="value" %>` The JSP directives give the JSP engine information about the JSP page. For example, `<%@ page import="java.util.*, java.text.*" %>` imports `java.util.*` and `java.text.*`.
- `<jsp:useBean id="objectName" scope="scopeAttribute" class="ClassName" />` Creates a bean if new. If a bean is already created, associates the id with the bean in the same scope.
- `<jsp:useBean id="objectName" scope="scopeAttribute" class="ClassName" > statements </jsp:useBean>` The statements are executed when the bean is created. If a bean with the same id and class name already exists, the statements are not executed.
- `<jsp:getProperty name="beanId" property="sample" />` Gets the property value from the bean, which is the same as `<%= beanId.getSample() %>`.
- `<jsp:setProperty name="beanId" property="sample" value="test1" />` Sets the property value for the bean, which is the same as `<% beanId.setSample("test1"); %>`.
- `<jsp:setProperty name="beanId" property="score" param="score" />` Sets the property with an input parameter.

- `<jsp:setProperty name="beanId" property="*" />` Associates and sets all the bean properties in `beanId` with the input parameters that match the property names.
- `<jsp:forward page="destination" />` Forwards this page to a new page.

JSP Predefined Variables

- `application` represents the `ServletContext` object for storing persistent data for all clients.
- `config` represents the `ServletConfig` object for the page.
- `out` represents the character output stream, which is an instance of `PrintWriter`, obtained from `response.getWriter()`.
- `page` is alternative to `this`.
- `request` represents the client's request, which is an instance of `HttpServletRequest` in the servlet's `service` method.
- `response` represents the client's response, which is an instance of `HttpServletResponse` in the servlet's `service` method.
- `session` represents the `HttpSession` object associated with the request, obtained from `request.getSession()`.

CHAPTER SUMMARY

1. A JavaServer page is like a regular HTML page with special tags, known as *JSP tags*, which enable the Web server to generate dynamic content. You can create a webpage with static HTML and enclose the code for generating dynamic content in the JSP tags.

2. A JSP page must be stored in a file with a .jsp extension. The Web server translates the .jsp file into a Java servlet, compiles the servlet, and executes it. The result of the execution is sent to the browser for display.

3. A JSP page is translated into a servlet when the page is requested for the first time. It is not retranslated if the page is not modified. To ensure that the first-time real user does not encounter a delay, JSP developers should test the page after it is installed.

4. There are three main types of JSP constructs: scripting constructs, directives, and actions. *Scripting* elements enable you to specify Java code that will become part of the resultant servlet. *Directives* enable you to control the overall structure of the resultant servlet. *Actions* enable you to control the behaviors of the JSP engine.

5. Three types of scripting constructs can be used to insert Java code into the resultant servlet: expressions, scriptlets, and declarations.

6. The scope attribute (application, session, page, and request) specifies the scope of a JavaBeans object. Application specifies that the object be bound to the application. Session specifies that the object be bound to the client's session. Page is the default scope, which specifies that the object be bound to the page. Request specifies that the object be bound to the client's request.

7. Web applications developed using JSP generally consist of many pages linked together. JSP provides a forwarding tag in the following syntax that can be used to forward a page to another page: `<jsp:forward page="destination" />`.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

Note
Solutions to even-numbered exercises in this chapter are in `exercise\jspexercise` from evennumberedexercise.zip, which can be downloaded from the Companion Website.

Section 38.4

38.1 (*Factorial table in JSP*) Rewrite Exercise 37.1 using JSP.

38.2 (*Muliplication table in JSP*) Rewrite Exercise 37.2 using JSP.

Section 38.5

***38.3** (*Obtain parameters in JSP*) Rewrite the servlet in Listing 37.4, GetParameters.java, using JSP. Create an HTML form that is identical to Student_Registration_Form.html in Listing 37.3 except that the action is replaced by `Exercise40_3.jsp` for obtaining parameter values.

Section 38.6

38.4 (*Calculate tax in JSP*) Rewrite Exercise 37.4 using JSP. You need to import ComputeTax in the JSP.

***38.5** (*Find scores from text files*) Rewrite Exercise 37.6 using servlets.

****38.6** (*Find scores from database tables*) Rewrite Exercise 37.7 using servlets.

Section 38.7

****38.7** (*Change the password*) Rewrite Exercise 37.8 using servlets.

Comprehensive

***38.8** (*Store cookies in JSP*) Rewrite Exercise 37.10 using JSP. Use `response.addCookie(Cookie)` to add a cookie.

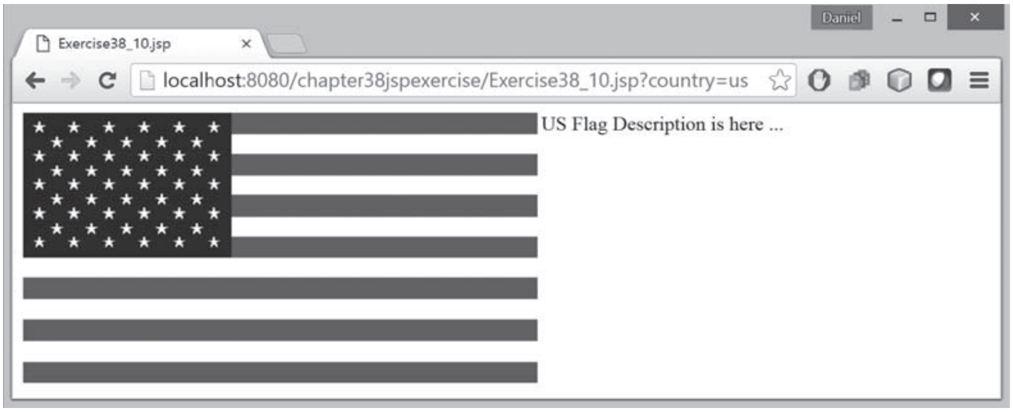
***38.9** (*Retrieve cookies in JSP*) Rewrite Exercise 37.11 using JSP. Use `Cookie[] cookies = request.getCookies()` to get all cookies.

38.10 (*Draw images*) Write a JSP program that displays a country's flag and description as shown in Figure 38.12. The country code such as us is passed as a parameter in the URL. The country's flag file is named as CountryCode.gif and the description is stored in a text file named CountryCode.txt on the server. So, for the country code us, the flag file us.gif and the text file is us.txt.

*****38.11** (*Syntax highlighting*) Rewrite Exercise 37.12 using JSP.

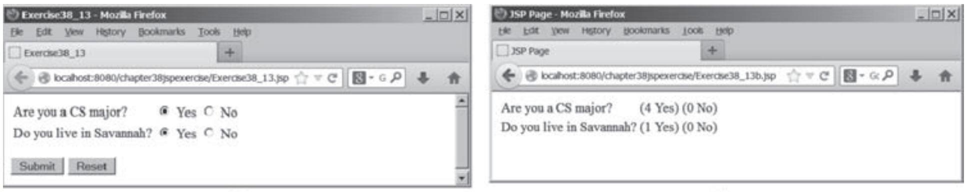
****38.12** (*Opinion poll*) Rewrite Exercise 37.13 using JSP.

*****38.13** (*Multiple-question opinion poll*) The `Poll` table in Exercise 37.13 contains only one question. Suppose you have a `Poll` table that contains multiple questions. Write a JSP that reads all the questions from the table and display them in a form, as shown in Figure 38.13a. When the user clicks the *Submit* button, another JSP page is invoked. This page updates the Yes or No counts for each question and displays the current Yes and No counts for each question in the `Poll` table, as shown in Figure 38.13b. Note that the table may contain many questions. The questions in the figure are just examples. Sort the questions in alphabetical order.


Exercise38_10.jsp
localhost:8080/chapter38jspexercise/Exercise38_10.jsp?country=us
Daniel

US Flag Description is here ...

FIGURE 38.12 The program displays an image and the description of the image.


Exercise38_13 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise38_13
localhost:8080/chapter38jspexercise/Exercise38_13.jsp
JSP Page - Mozilla Firefox
JSP Page
localhost:8080/chapter38jspexercise/Exercise38_13b.jsp

Are you a CS major? ● Yes ○ No
Do you live in Savannah? ● Yes ○ No

Submit Reset

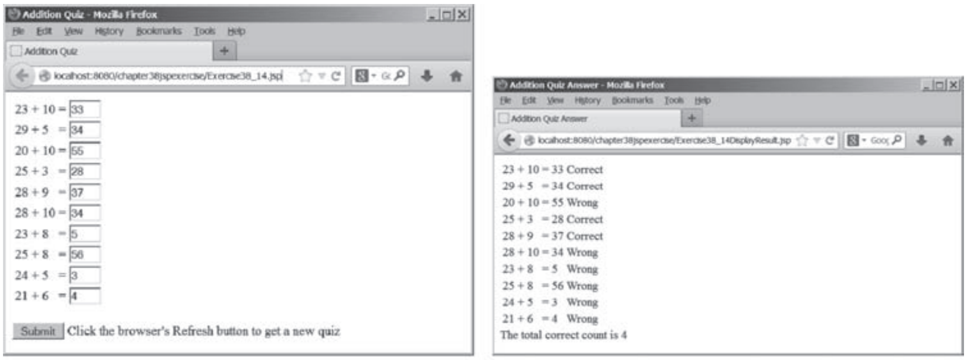
(a)

Are you a CS major? (4 Yes) (0 No)
Do you live in Savannah? (1 Yes) (0 No)

(b)

FIGURE 38.13 The form prompts the user to enter Yes or No for each question in (a), and the updated Yes or No counts are displayed in (b).

****38.14** (*Addition quiz*) Write a JSP program that generates addition quizzes randomly, as shown in Figure 38.14a. After the user answers all questions, the JSP displays the result, as shown in Figure 38.14b.


Addition Quiz - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Addition Quiz
localhost:8080/chapter38jspexercise/Exercise38_14.jsp
Addition Quiz Answer - Mozilla Firefox
Addition Quiz Answer
localhost:8080/chapter38jspexercise/Exercise38_14DisplayResult.jsp

```
23 + 10 = 33
29 + 5  = 34
20 + 10 = 55
25 + 3  = 28
28 + 9  = 37
28 + 10 = 34
23 + 8  = 5
25 + 8  = 56
24 + 5  = 3
21 + 6  = 4

Submit Click the browser's Refresh button to get a new quiz
```

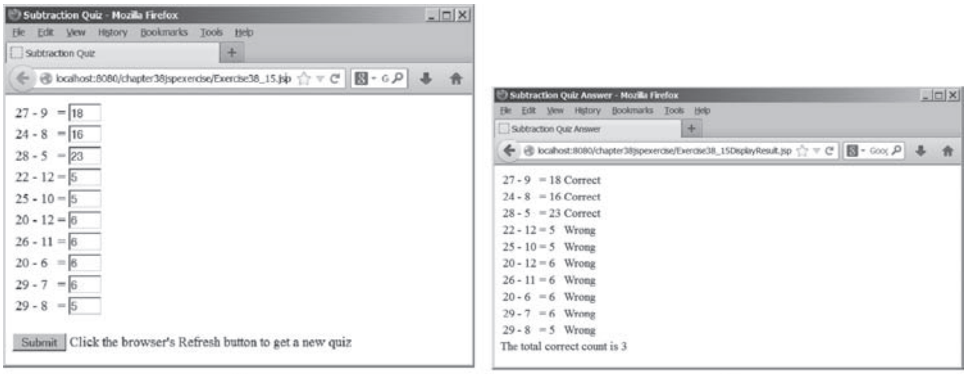
(a)

```
23 + 10 = 33 Correct
29 + 5  = 34 Correct
20 + 10 = 55 Wrong
25 + 3  = 28 Correct
28 + 9  = 37 Correct
28 + 10 = 34 Wrong
23 + 8  = 5  Wrong
25 + 8  = 56 Wrong
24 + 5  = 3  Wrong
21 + 6  = 4  Wrong
The total correct count is 4
```

(b)

FIGURE 38.14 The program displays addition questions in (a) and answers in (b).

****38.15** (*Subtraction quiz*) Write a JSP program that generates subtraction quizzes randomly, as shown in Figure 38.14a. The first number must always be greater than or equal to the second number. After the user answers all questions, the JSP displays the result, as shown in Figure 38.14b.


Subtraction Quiz - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Subtraction Quiz
localhost:8080/chapter38jspexercise/Exercise38_15.jsp
Submit
Subtraction Quiz Answer - Mozilla Firefox
Subtraction Quiz Answer
localhost:8080/chapter38jspexercise/Exercise38_15DisplayResult.jsp

```
27 - 9  = 18
24 - 8  = 16
28 - 5  = 23
22 - 12 = 5
25 - 10 = 5
20 - 12 = 6
26 - 11 = 6
20 - 6  = 6
29 - 7  = 6
29 - 8  = 5

Submit Click the browser's Refresh button to get a new quiz
```

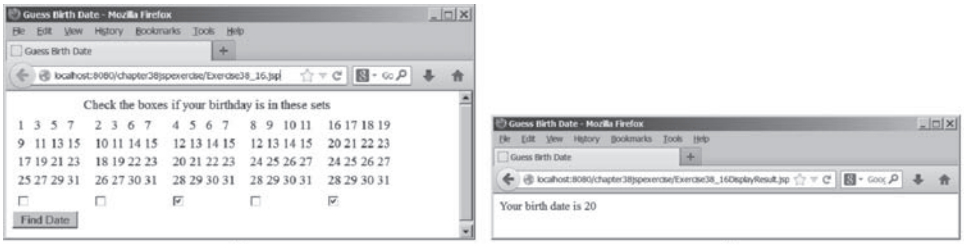
(a)

```
27 - 9  = 18 Correct
24 - 8  = 16 Correct
28 - 5  = 23 Correct
22 - 12 = 5  Wrong
25 - 10 = 5  Wrong
20 - 12 = 6  Wrong
26 - 11 = 6  Wrong
20 - 6  = 6  Wrong
29 - 7  = 6  Wrong
29 - 8  = 5  Wrong
The total correct count is 3
```

(b)

FIGURE 38.14 The program displays subtraction questions in (a) and answers in (b).

****38.16** (*Guess birthday*) Listing 3.3, GuessBirthDay.java, gives a program for guessing a birthday. Write a JSP program that displays five sets of numbers, as shown in Figure 38.15a. After the user checks the appropriate boxes and clicks the *Find Date* button, the program displays the date, as shown in Figure 38.15b.


Guess Birth Date - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Guess Birth Date
localhost:8080/chapter38jspexercise/Exercise38_16.jsp
Find Date
Guess Birth Date - Mozilla Firefox
Guess Birth Date
localhost:8080/chapter38jspexercise/Exercise38_16DisplayResult.jsp

```
            Check the boxes if your birthday is in these sets
1  3  5  7    2  3  6  7    4  5  6  7    8  9  10 11   16 17 18 19
9  11 13 15   10 11 14 15   12 13 14 15   12 13 14 15   20 21 22 23
17 19 21 23   18 19 22 23   20 21 22 23   24 25 26 27   24 25 26 27
25 27 29 31   26 27 30 31   28 29 30 31   28 29 30 31   28 29 30 31
☐             ☐             ☑             ☐             ☑
Find Date
```

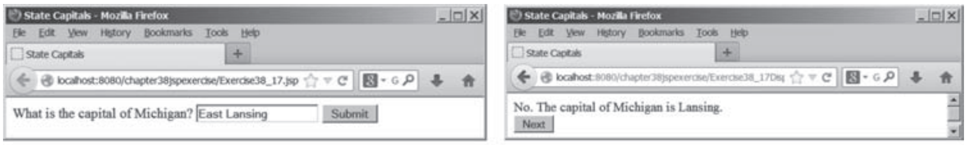
(a)

```
Your birth date is 20
```

(b)

FIGURE 38.15 (a) The program displays five sets of numbers for the user to check the boxes. (b) The program displays the date.

****38.17** (*Guess capitals*) Write a JSP that prompts the user to enter a capital for a state, as shown in Figure 38.16a. Upon receiving the user input, the program reports whether the answer is correct, as shown in Figure 38.16b. You can click the *Next* button to display another question. You can use a two-dimensional array to store the states and capitals, as proposed in Exercise 9.22. Create a list from the array and apply the `shuffle` method to reorder the list so the questions will appear in random order.


State Capitals - Mozilla Firefox
File Edit View History Bookmarks Tools Help
State Capitals
localhost:8080/chapter38jspexercise/Exercise38_17.jsp
Submit
State Capitals - Mozilla Firefox
State Capitals
localhost:8080/chapter38jspexercise/Exercise38_17Dis
Next

```
What is the capital of Michigan? East Lansing
```

(a)

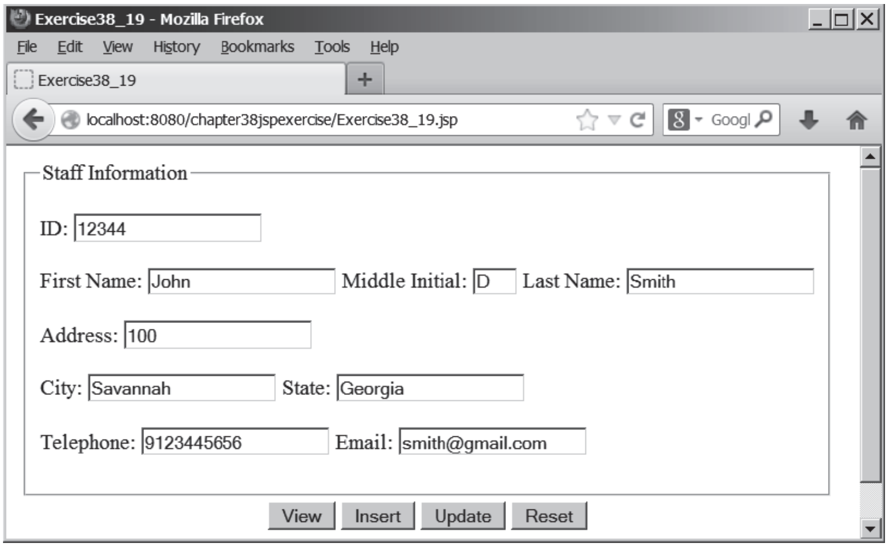
```
No. The capital of Michigan is Lansing.
```

(b)

FIGURE 38.16 (a) The program displays a question. (b) The program displays the answer to the question.

***38.18** (*Large factorial*) Rewrite Listing 38.11 to handle a large factorial. Use the `BigInteger` class introduced in §14.12.

****38.19** (*Access and update a* `Staff` *table*) Write a JSP for Exercise 33.1, as shown in Figure 38.17.


Exercise38_19 - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
Exercise38_19
localhost:8080/chapter38jspexercise/Exercise38_19.jsp
Googl
View
Insert
Update
Reset

Staff Information

ID: 12344

First Name: John Middle Initial: D Last Name: Smith

Address: 100

City: Savannah State: Georgia

Telephone: 9123445656 Email: smith@gmail.com

FIGURE 38.17 The JSP page lets you view, insert, and update staff information.

***38.20** (*Guess number*) Write a JSP page that generates a random number between `1` and `1000` and let the user enter a guess. When the user enters a guess, the program should tell the user whether the guess is correct, too high, or too low.

CHAPTER

39

JAVASERVER FACES

Objectives

- To explain what JSF is (§39.1).
- To create a JSF project in NetBeans (§39.2.1).
- To create a JSF page (§39.2.2).
- To create a JSF managed bean (§39.2.3).
- To use JSF expressions in a facelet (§39.2.4).
- To use JSF GUI components (§39.3).
- To obtain and process input from a form (§39.4).
- To develop a calculator using JSF (§39.5).
- To track sessions in application, session, view, and request scopes (§39.6).
- To validate input using the JSF validators (§39.7).
- To bind database with facelets (§39.8).
- To open a new JSF page from the current page (§39.9).
- To program using contexts and dependency injection (§39.10).

39.1 Introduction

JavaServer Faces (JSF) is a new technology for developing server-side Web applications using Java.

JSF

JSF enables you to completely separate Java code from HTML. You can quickly build Web applications by assembling reusable UI components in a page, connecting these components to Java programs and wiring client-generated events to server-side event handlers. The application developed using JSF is easy to debug and maintain.

JSF 2
XHTML
CSS

Note

This chapter introduces JSF 2, the latest standard for JavaServer Faces. You need to know XHTML (eXtensible HyperText Markup Language) and CSS (Cascading Style Sheet) to start this chapter. For information on XHTML and CSS, see Supplements V.A and V.B on the Companion Website.

NetBeans 7.3.1
GlassFish 4
Java EE 7

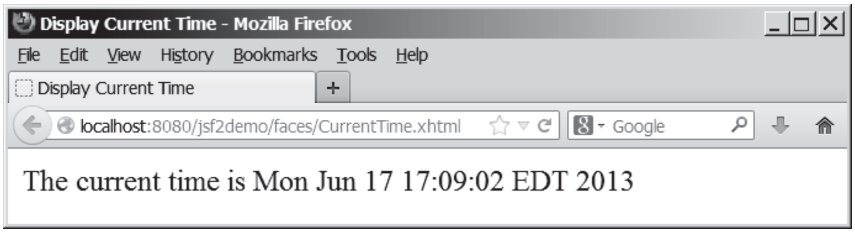
Caution

The examples and exercises in this chapter were tested using NetBeans 7.3.1, GlassFish 4, and Java EE 7. You need to use NetBeans 7.3.1 or a higher version with GlassFish 4 and Java EE 7 to develop your JSF projects.

39.2 Getting Started with JSF

NetBeans is an effective tool for developing JSF applications.

We begin with a simple example that illustrates the basics of developing JSF projects using NetBeans. The example is to display the date and time on the server, as shown in Figure 39.1.



localhost:8080/jsf2demo/faces/CurrentTime.xhtml

The current time is Mon Jun 17 17:09:02 EDT 2013

FIGURE 39.1 The application displays the date and time on the server.

39.2.1 Creating a JSF Project

Here are the steps to create the application.

create a project

Step 1: Choose ***File*, *New Project*** to display the New Project dialog box. In this box, choose *Java Web* in the Categories pane and *Web Application* in the Projects pane. Click *Next* to display the New Web Application dialog box.

In the New Web Application dialog box, enter and select the following fields, as shown in Figure 39.2a:

Project Name: `jsf2demo`
Project Location: `c:\book`

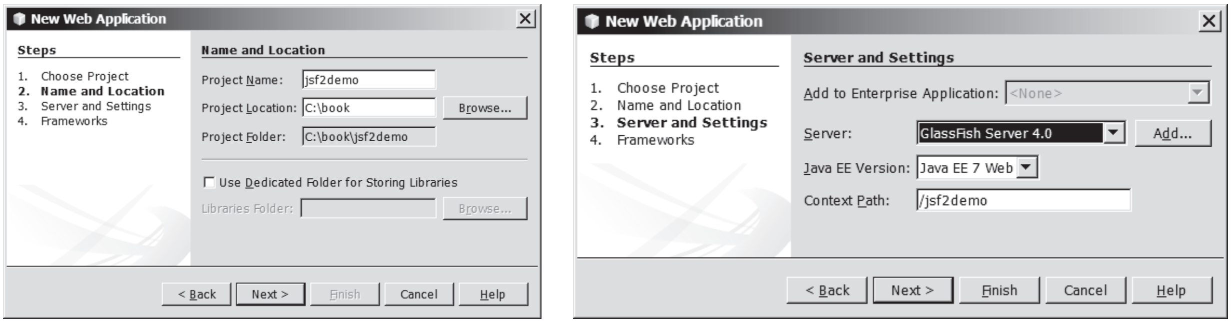
choose server and Java EE 7

Step 2: Click *Next* to display the dialog box for choosing servers and settings. Select the following fields as shown in Figure 39.2b. (Note: You can use any server such as GlassFish 4.x that supports Java EE 6.)

Server: `GlassFish 4`
Java EE Version: `Java EE 7 Web`

Step 3: Click *Next* to display the dialog box for choosing frameworks, as shown in Figure 39.3. Check *JavaServer Faces* and JSF 2.0 as Server Library. Click *Finish* to create the project, as shown in Figure 39.4.

choose JavaServer Faces and JSF2.2


New Web Application
Steps
1. Choose Project
2. Name and Location
3. Server and Settings
4. Frameworks
Browse...
< Back
Next >
Finish
Cancel
Help
Add...

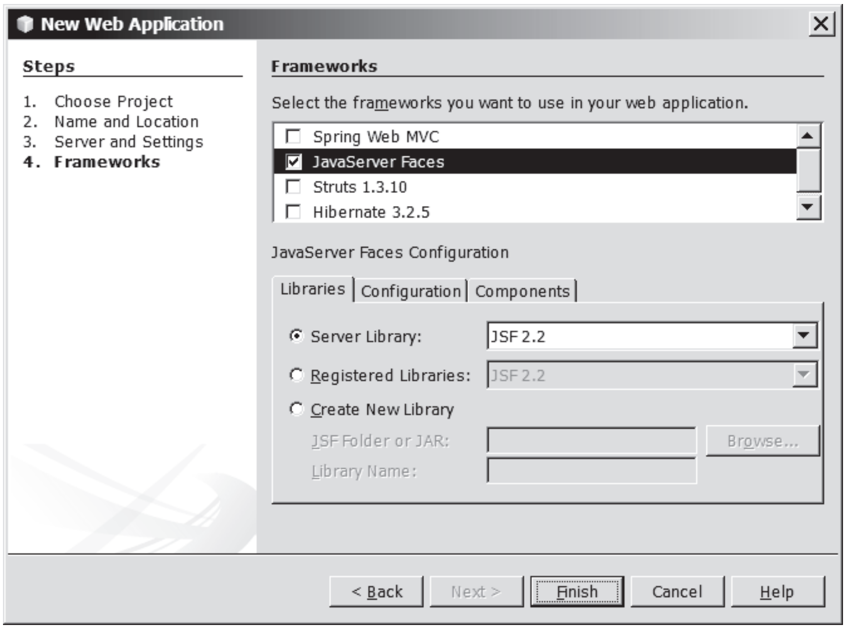
Name and Location
Project Name: jsf2demo
Project Location: C:\book
Project Folder: C:\book\jsf2demo
☐ Use Dedicated Folder for Storing Libraries
Libraries Folder:

(a)

Server and Settings
Add to Enterprise Application: <None>
Server: GlassFish Server 4.0
Java EE Version: Java EE 7 Web
Context Path: /jsf2demo

(b)

FIGURE 39.2 The New Web Application dialog box enables you to create a new Web project.


New Web Application
Steps
1. Choose Project
2. Name and Location
3. Server and Settings
4. Frameworks
Libraries
Configuration
Components
Browse...
< Back
Next >
Finish
Cancel
Help

Frameworks

Select the frameworks you want to use in your web application.

☐ Spring Web MVC
☑ JavaServer Faces
☐ Struts 1.3.10
☐ Hibernate 3.2.5

JavaServer Faces Configuration

● Server Library: JSF 2.2
○ Registered Libraries: JSF 2.2
○ Create New Library
JSF Folder or JAR:
Library Name:

FIGURE 39.3 Check JavaServer Faces and JSF 2.2 to create a Web project.

39.2.2 A Basic JSF Page

A new project was just created with a default page named index.xhtml, as shown in Figure 39.4. This page is known as a *facelet*, which mixes JSF tags with XHTML tags. Listing 39.1 lists the contents of index.xhtml.

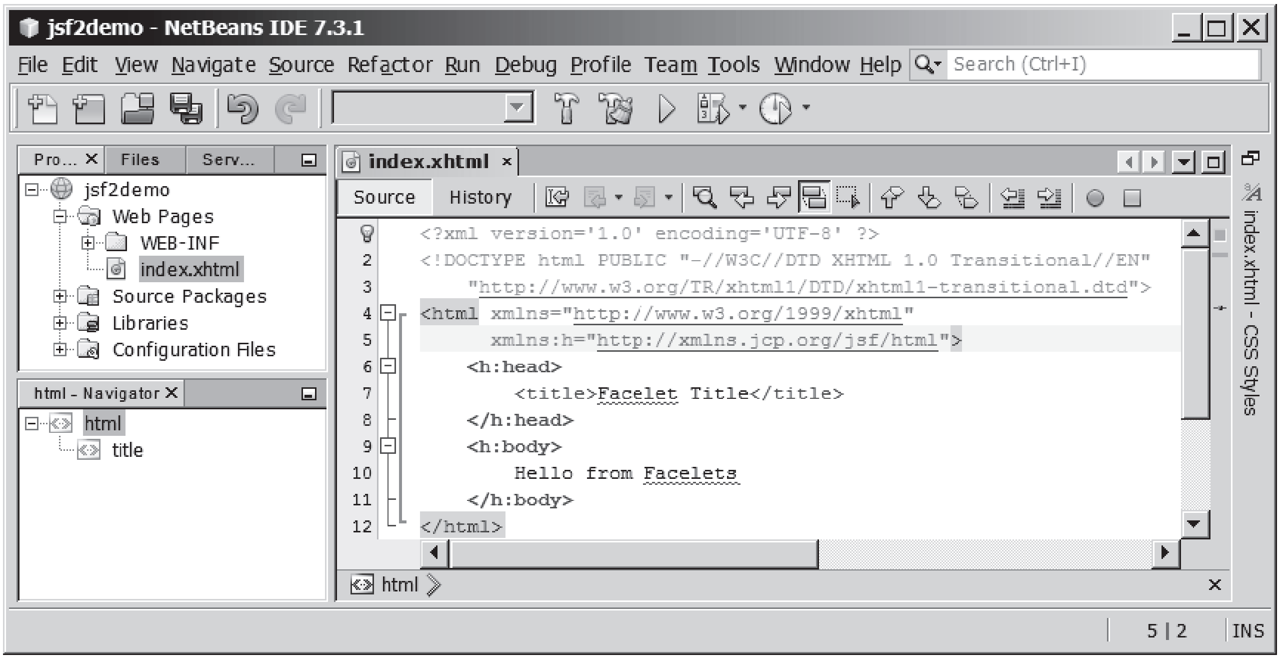
facelet

LISTING 39.1 `index.xhtml`

```
<?xml version='1.0' encoding='UTF-8' ?>
<!-- index.xhtml -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
```

xml version
comment
DOCTYPE

default namespace
JSF namespace


jsf2demo - NetBeans IDE 7.3.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
Pro... Files Serv...
jsf2demo
Web Pages
WEB-INF
index.xhtml
Source Packages
Libraries
Configuration Files
html - Navigator
html
title
index.xhtml
Source History
index.xhtml - CSS Styles
html
5 | 2
INS

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Facelet Title</title>
    </h:head>
    <h:body>
        Hello from Facelets
    </h:body>
</html>
```

FIGURE 39.4 A default JSF page is created in a new Web project.

h:head

h:body

```
 7    <h:head>
 8      <title>Facelet Title</title>
 9    </h:head>
10    <h:body>
11      Hello from Facelets
12    </h:body>
13  </html>
```

XML declaration

Line 1 is an XML declaration to state that the document conforms to the XML version 1.0 and uses the UTF-8 encoding. The declaration is optional, but it is a good practice to use it. A document without the declaration may be assumed of a different version, which may lead to errors. If an XML declaration is present, it must be the first item to appear in the document. This is because an XML processor looks for the first line to obtain information about the document so that it can be processed correctly.

XML comment

Line 2 is a comment for documenting the contents in the file. XML comment always begins with `<!--` and ends with `-->`.

DOCTYPE

Lines 3 and 4 specify the version of XHTML used in the document. This can be used by the Web browser to validate the syntax of the document.

element
tag

An XML document consists of elements described by tags. An element is enclosed between a start tag and an end tag. XML elements are organized in a tree-like hierarchy. Elements may contain subelements, but there is only one root element in an XML document. All the elements must be enclosed inside the root tag. The root element in XHTML is defined using the `html` tag (line 5).

Each tag in XML must be used in a pair of the start tag and the end tag. A start tag begins with `<` followed by the tag name and ends with `>`. An end tag is the same as its start tag except that it begins with `</`. The start tag and end tag for `html` are `<html>` and `</html>`.

html tag

The `html` element is the root element that contains all other elements in an XHTML page. The starting `<html>` tag (lines 5 and 6) may contain one or more `xmlns` (XML namespace) attributes to specify the namespace for the elements used in the document. Namespaces are like Java packages. Java packages are used to organize classes and to avoid naming conflict. XHTML namespaces are used to organize tags and resolve naming conflict. If an element with the same name is defined in two namespaces, the fully qualified tag names can be used to differentiate them.

Each `xmlns` attribute has a name and a value separated by an equal sign (`=`). The following declaration (line 5)

`xmlns`

```
xmlns="http://www.w3.org/1999/xhtml"
```

specifies that any unqualified tag names are defined in the default standard XHTML namespace.

The following declaration (line 6)

```
xmlns:h="http://xmlns.jcp.org/jsf/html"
```

allows the tags defined in the JSF tag library to be used in the document. These tags must have a prefix `h`.

An `html` element contains a head and a body. The `h:head` element (lines 7–9) defines an HTML `title` element. The title is usually displayed in the browser window's title bar.

h:head

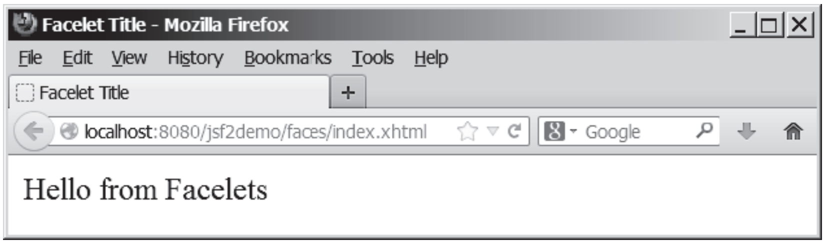
An `h:body` element defines the page's content. In this simple example, it contains a string to be displayed in the Web browser.

h:body

Note

The XML tag names are case sensitive, whereas HTML tags are not. So, `<html>` is different from `<HTML>` in XML. Every start tag in XML must have a matching end tag, whereas some tags in HTML do not need end tags.

You can now display the page in index.xhtml by right-clicking on index.xhtml in the projects pane and choose *Run File*. The page is displayed in a browser, as shown in Figure 39.5.



localhost:8080/jsf2demo/faces/index.xhtml

Hello from Facelets

Figure 39.5 The index.xhtml is displayed in the browser.

Note

The JSF page is processed and converted into a regular HTML page for displaying by a browser. The Java software that runs on the server side for producing the HTML page is known as *Java server container* or simply *container*. The container is responsible for handling all server-side tasks for Java EE. GlassFish is a Java server container.

container

39.2.3 Managed JavaBeans for JSF

JSF applications are developed using the Model-View-Controller (MVC) architecture, which separates the application's data (contained in the model) from the graphical presentation (the view). The controller is the JSF framework that is responsible for coordinating interactions between view and the model.

In JSF, the facelets are the view for presenting data. Data are obtained from Java objects. Objects are defined using Java classes. In JSF, the objects that are accessed from a facelet are JavaBeans objects. A *JavaBean* class is simply a public Java class with a no-arg constructor. JavaBeans may contain properties. By convention, a property is defined with a getter and a setter method. If a property only has a getter method, the property is called a read-only property. If a property only has a setter method, the property is called a write-only property. A property does not need to be defined as a data field in the class.

JavaBean

Our example in this section is to develop a JSF facelet to display current time. We will create a JavaBean with a `getTime()` method that returns the current time as a string. The facelet will invoke this method to obtain current time.

Here are the steps to create a JavaBean named `TimeBean`.

Step 1. Right-click the project node jsf2demo to display a context menu as shown in Figure 39.6. Choose *New*, *JSF Managed Bean* to display the New JSF Managed Bean dialog box, as shown in Figure 39.7. (Note: if you don't see JSF Managed Bean in the menu, choose *Other* to locate it in the JavaServer Faces category.)

Step 2. Enter and select the following fields, as shown in Figure 39.7:

Class Name: `TimeBean`
Package: `jsf2demo`
Name: `timeBean`
Scope: `request`
Click *Finish* to create TimeBean.java, as shown in Figure 39.8.

Step 3. Add the `getTime()` method to return the current time, as shown in Listing 39.2.

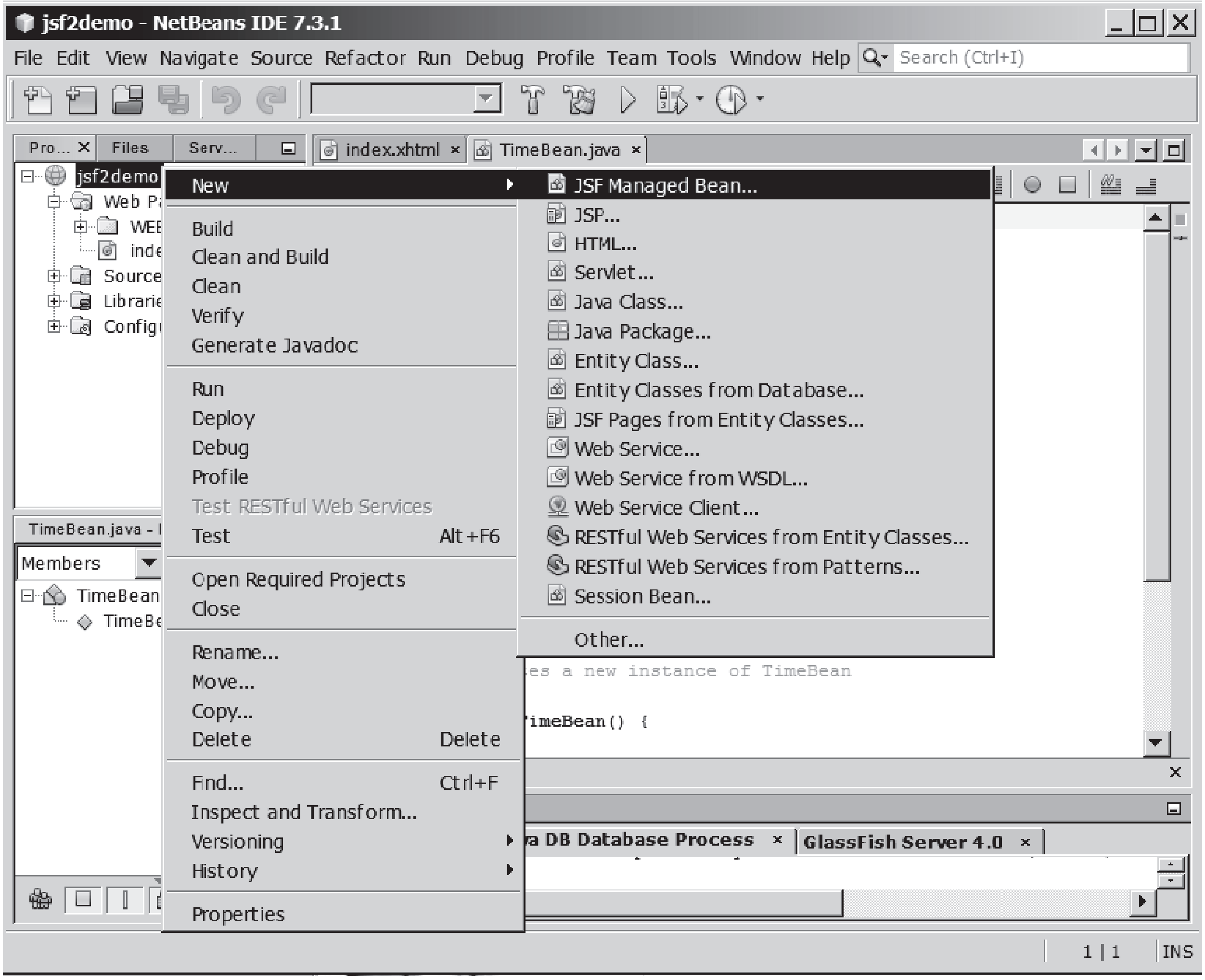
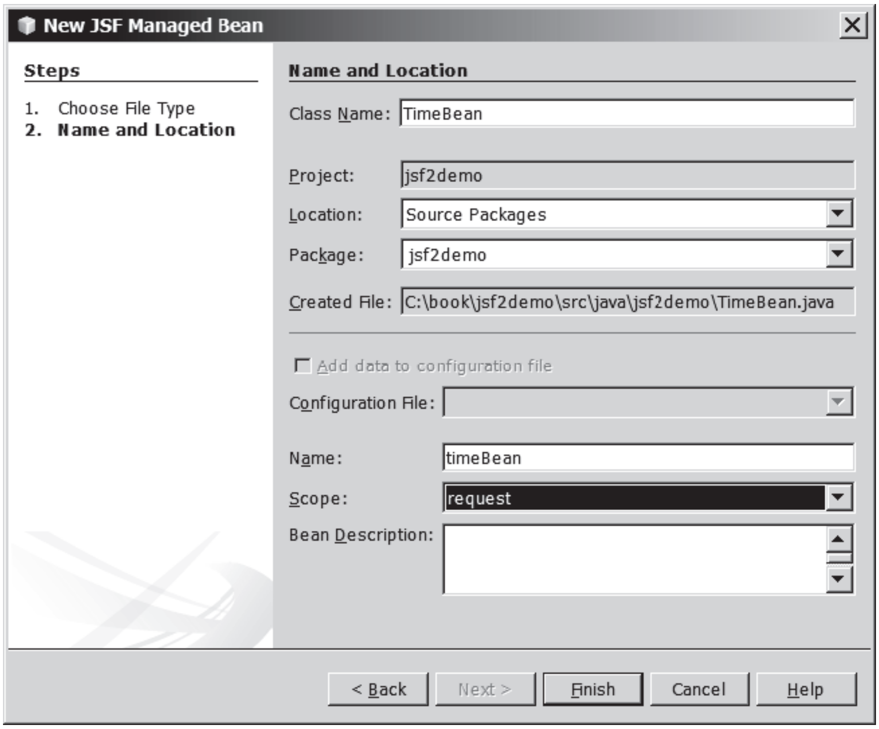

jsf2demo - NetBeans IDE 7.3.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
Pro... Files Serv...
index.xhtml
TimeBean.java
jsf2demo
Web Pa
WEB
inde
Source
Librarie
Configu
TimeBean.java - I
Members
TimeBean
TimeBe
New
Build
Clean and Build
Clean
Verify
Generate Javadoc
Run
Deploy
Debug
Profile
Test RESTful Web Services
Test Alt+F6
Open Required Projects
Close
Rename...
Move...
Copy...
Delete Delete
Find... Ctrl+F
Inspect and Transform...
Versioning
History
Properties
JSF Managed Bean...
JSP...
HTML...
Servlet...
Java Class...
Java Package...
Entity Class...
Entity Classes from Database...
JSF Pages from Entity Classes...
Web Service...
Web Service from WSDL...
Web Service Client...
RESTful Web Services from Entity Classes...
RESTful Web Services from Patterns...
Session Bean...
Other...
es a new instance of TimeBean
imeBean() {
a DB Database Process
GlassFish Server 4.0
1 | 1
INS

FIGURE 39.6 Choose JSF Managed Bean to create a JavaBean for JSF.



New JSF Managed Bean

Steps

1. Choose File Type
2. **Name and Location**

Name and Location

Class Name: TimeBean

Project: jsf2demo

Location: Source Packages

Package: jsf2demo

Created File: C:\book\jsf2demo\src\java\jsf2demo\TimeBean.java

☐ Add data to configuration file

Configuration File:

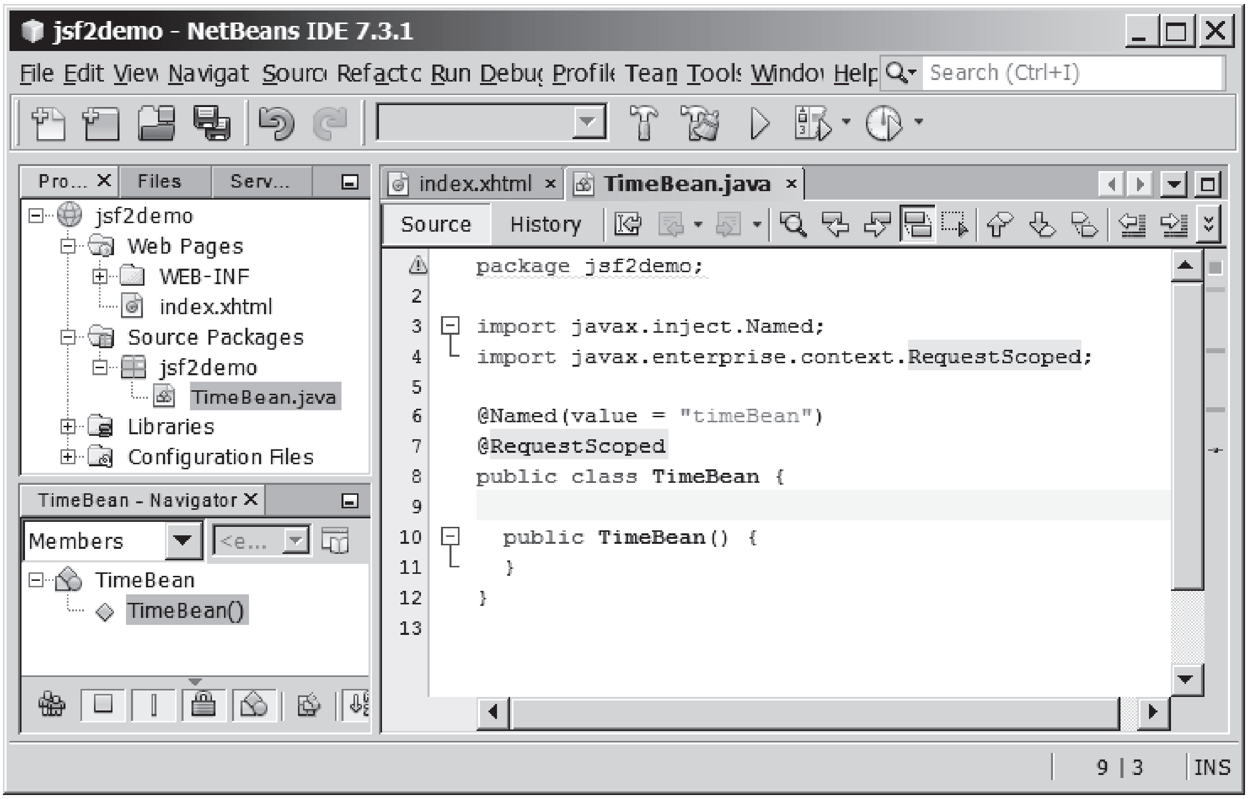
Name: timeBean

Scope: request

Bean Description:

< Back
Next >
Finish
Cancel
Help

FIGURE 39.7 Specify the name, location, and scope for the bean.



jsf2demo - NetBeans IDE 7.3.1

File Edit View Navigat Sourc Refactc Run Debu Profil Tean Tools Windo Help
Search (Ctrl+I)
Pro... Files Serv...
jsf2demo
Web Pages
WEB-INF
index.xhtml
Source Packages
jsf2demo
TimeBean.java
Libraries
Configuration Files
TimeBean - Navigator
Members
<e...
TimeBean
TimeBean()
index.xhtml
TimeBean.java
Source
History
9 | 3
INS

```
package jsf2demo;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

@Named(value = "timeBean")
@RequestScoped
public class TimeBean {

  public TimeBean() {
  }
}

```

FIGURE 39.8 A JavaBean for JSF was created.

LISTING 39.2 `TimeBean.java`

@Named
@RequestScoped
time property

```java
package jsf2demo;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

@Named (value = "timeBean")
@RequestScoped
public class TimeBean {
  public TimeBean() {
  }

  public String getTime() {
    return new java.util.Date().toString();
  }
}
```

`TimeBean` is a JavaBeans with the `@Named` annotation, which indicates that the JSF framework will create and manage the `TimeBean` objects used in the application. You have learned to use the `@Override` annotation in Chapter 11. The `@Override` annotation tells the compiler that the annotated method is required to override a method in a superclass. The `@Named` annotation tells the compiler to generate the code to enable the bean to be used by JSF facelets.

@RequestScope

The `@RequestScope` annotation specifies that the scope of the JavaBeans object is within a request. You can also use `@ViewScope`, `@SessionScope` or `@ApplicationScope` to specify the scope for a session or for the entire application.

39.2.4 JSF Expressions

We demonstrate JSF expressions by writing a simple application that displays the current time. You can display current time by invoking the `getTime()` method in a `TimeBean` object using a JSF expression.

To keep index.xhtml intact, we create a new JSF page named CurrentTime.xhtml as follows:

Step 1. Right-click the `jsf2demo` node in the project pane to display a context menu and choose *New*, *JSF Page* to display the New JSF File dialog box, as shown in Figure 39.9.

Step 2. Enter `CurrentTime` in the File Name field, choose Facelets and click *Finish* to generate CurrentTime.xhtml, as shown in Figure 39.10.

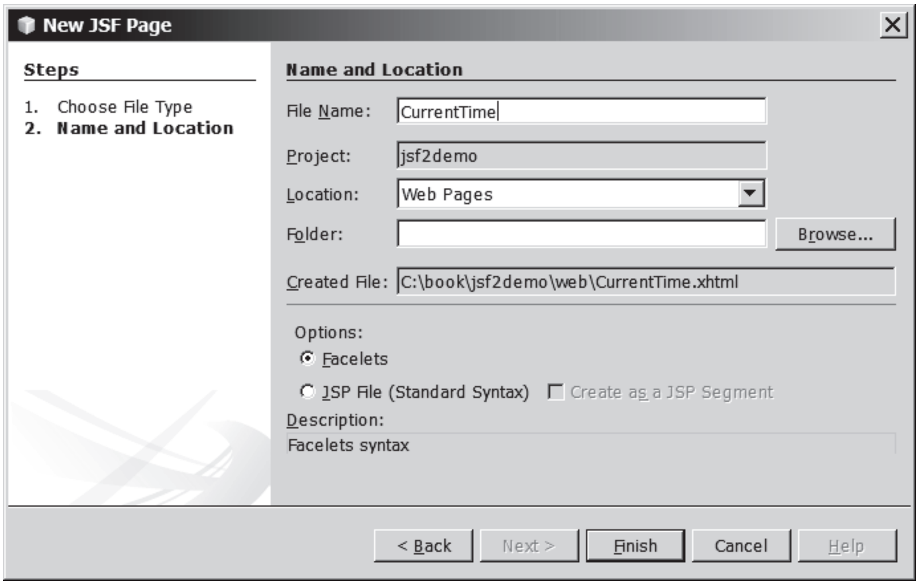
Step 3. Add a JSF expression to obtain the current time, as shown in Listing 39.3.

Step 4. Right-click on CurrentTime.xhtml in the project to display a context menu and choose *Run File* to display the page in a browser as shown in Figure 39.1.

LISTING 39.3 `CurrentTime.xhtml`

refresh page
JSF expression

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
     <title>Display Current Time</title>
     <meta http-equiv="refresh" content ="60" />
  </h:head>
  <h:body>
    The current time is #{timeBean.time}
  </h:body>
</html>
```



New JSF Page
Steps
1. Choose File Type
2. Name and Location
< Back
Next >
Finish
Cancel
Help
Browse...

Name and Location

File Name: CurrentTime

Project: jsf2demo

Location: Web Pages

Folder:

Created File: C:\book\jsf2demo\web\CurrentTime.xhtml

Options:

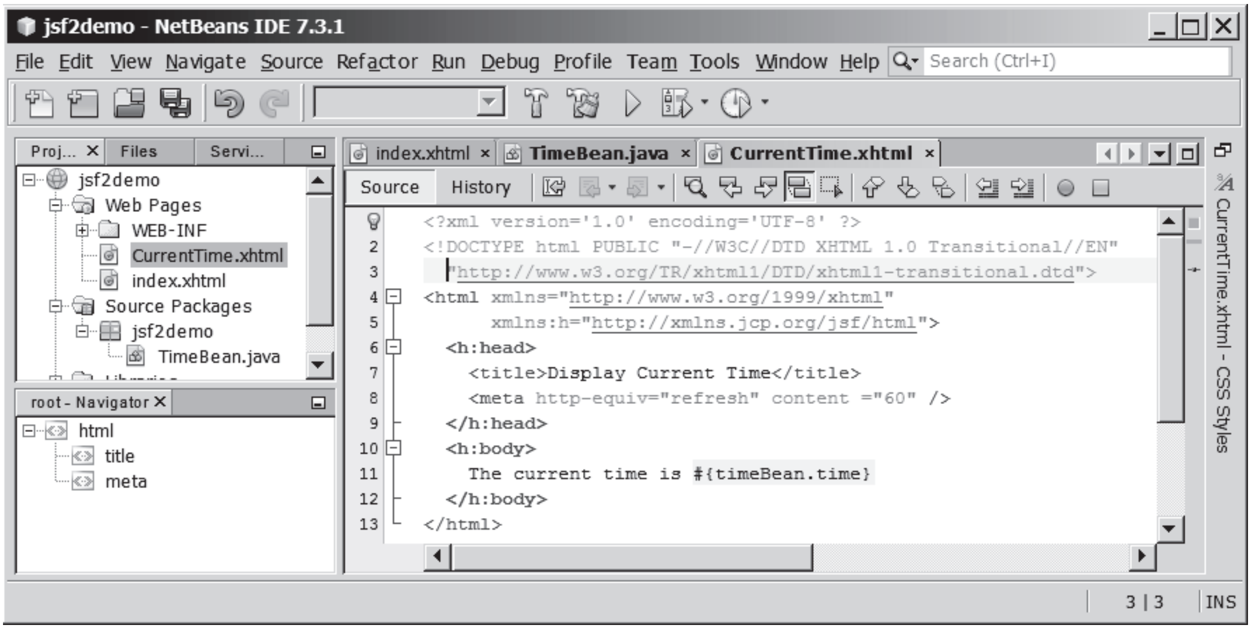
● Facelets

○ JSP File (Standard Syntax) ☐ Create as a JSP Segment

Description:

Facelets syntax

FIGURE 39.9 The New JSF Page dialog is used to create a JSF page.



jsf2demo - NetBeans IDE 7.3.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
Proj... Files Servi...
jsf2demo
Web Pages
WEB-INF
CurrentTime.xhtml
index.xhtml
Source Packages
jsf2demo
TimeBean.java
root - Navigator
html
title
meta
index.xhtml
TimeBean.java
CurrentTime.xhtml
Source
History
CurrentTime.xhtml - CSS Styles
3 | 3
INS

```
1   <?xml version='1.0' encoding='UTF-8' ?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4   <html xmlns="http://www.w3.org/1999/xhtml"
5         xmlns:h="http://xmlns.jcp.org/jsf/html">
6     <h:head>
7       <title>Display Current Time</title>
8       <meta http-equiv="refresh" content ="60" />
9     </h:head>
10    <h:body>
11      The current time is #{timeBean.time}
12    </h:body>
13  </html>
```

FIGURE 39.10 A New JSF page CurrentTime was created.

Line 11 defines a `meta` tag inside the `h:head` tag to tell the browser to refresh every 60 seconds. This line can also be written as

```
<meta http-equiv="refresh" content ="60"></ meta>
```

An element is called an *empty element* if there are no contents between the start tag and the end tag. In an empty element, data are typically specified as attributes in the start tag. You can close an empty element by placing a slash immediately preceding the start tag's right angle bracket, as shown in line 8, for brevity.

empty element

Line 8 uses a JSF expression `#{timeBean.time}` to obtain the current time. `timeBean` is an object of the `TimeBean` class. The object name can be changed in the `@Named` annotation (line 6 in Listing 39.2) using the following syntax:

```
@Named(name = "anyObjectName")
```

By default, the object name is the class name with the first letter in lowercase.

Note that `time` is a JavaBeans property because the `getTime()` method is defined in TimeBeans. The JSF expression can either use the property name or invoke the method to obtain the current time. So the following two expressions are fine.

```
#{timeBean.time}
#{timeBean.getTime()}
```

The syntax of a JSF expression is

```
#{expression}
```

JSF expressions bind JavaBeans objects with facelets. You will see more use of JSF expressions in the upcoming examples in this chapter.

Check Point

39.2.1 What is JSF?

39.2.2 How do you create a JSF project in NetBeans?

39.2.3 How do you create a JSF page in a JSF project?

39.2.4 What is a facelet?

39.2.5 What is the file extension name for a facelet?

39.2.6 What is a managed bean?

39.2.7 What is the `@Named` annotation for?

39.2.8 What is the `@RequestScope` annotation for?

39.3 JSF GUI Components

Key Point

JSF provides many elements for displaying GUI components.

Table 39.1 lists some of the commonly used elements. The tags with the `h` prefix are in the JSF HTML Tag library. The tags with the `f` prefix are in the JSF Core Tag library.

Listing 39.4 is an example that uses some of these elements to display a student registration form, as shown in Figure 39.11.

Listing 39.4 StudentRegistrationForm.xhtml

jsf core namespace

graphicImage

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
  <h:head>
    <title>Student Registration Form</title>
  </h:head>
  <h:body>
    <h:form>
      <!-- Use h:graphicImage -->
      <h3>Student Registration Form
         <h:graphicImage name="usIcon.gif" library="image"/>
      </h3>
```

TABLE 39.1 JSF GUI Form Elements

JSF Tag	*Description*
`h:form`	`inserts an XHTML form into a page.`
`h:panelGroup`	`similar to a JavaFX FlowPane.`
`h:panelGrid`	`similar to a JavaFX GridPane.`
`h:inputText`	`displays a textbox for entering input.`
`h:outputText`	`displays a textbox for displaying output.`
`h:inputTextArea`	`displays a textarea for entering input.`
`h:inputSecret`	`displays a textbox for entering password.`
`h:outputLabel`	`displays a label.`
`h:outputLink`	`displays a hypertext link.`
`h:selectOneMenu`	`displays a combo box for selecting one item.`
`h:selectOneRadio`	`displays a set of radio button.`
`h:selectManyCheckbox`	`displays checkboxes.`
`h:selectOneListbox`	`displays a list for selecting one item.`
`h:selectManyListbox`	`displays a list for selecting multiple items.`
`f:selectItem`	`specifies an item in an h:selectOneMenu, h:selectOneRadio, or h:selectManyListbox.`
`h:message`	`displays a message for validating input.`
`h:dataTable`	`displays a data table.`
`h:column`	`specifies a column in a data table.`
`h:graphicImage`	`displays an image.`

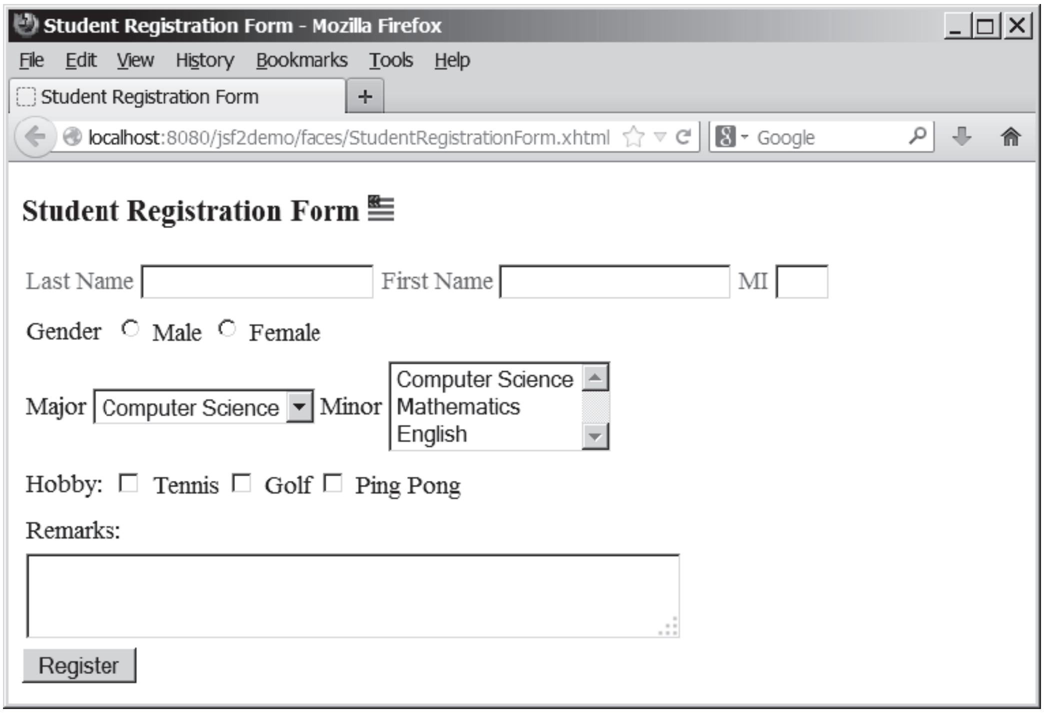

Student Registration Form - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Student Registration Form
localhost:8080/jsf2demo/faces/StudentRegistrationForm.xhtml
Google
Student Registration Form
Last Name
First Name
MI
Gender ○ Male ○ Female
Major Computer Science
Minor
Computer Science
Mathematics
English
Hobby: ☐ Tennis ☐ Golf ☐ Ping Pong
Remarks:
Register

FIGURE 39.11 A student registration form is displayed using JSF elements.

h:panelGrid
h:outputLabel
h:inputText
h:selectOneRadio
f:selectItem
h:selectOneMenu
h:selectManyListBox
h:selectManyCheckbox
h:inputTextarea
h:commandButton

```

      <!-- Use h:panelGrid -->
      <h:panelGrid columns="6" style="color:green">
        <h:outputLabel value="Last Name"/>
        <h:inputText id="lastNameInputText" />
        <h:outputLabel value="First Name" />
        <h:inputText id="firstNameInputText" />
        <h:outputLabel value="MI" />
        <h:inputText id="miInputText" size="1" />
      </h:panelGrid>

      <!-- Use radio buttons -->
      <h:panelGrid columns="2">
        <h:outputLabel>Gender </h:outputLabel>
        <h:selectOneRadio id="genderSelectOneRadio">
          <f:selectItem itemValue="Male"
                        itemLabel="Male"/>
          <f:selectItem itemValue="Female"
                        itemLabel="Female"/>
        </h:selectOneRadio>
      </h:panelGrid>

      <!-- Use combo box and list -->
      <h:panelGrid columns="4">
        <h:outputLabel value="Major "/>
        <h:selectOneMenu id="majorSelectOneMenu">
          <f:selectItem itemValue="Computer Science"/>
          <f:selectItem itemValue="Mathematics"/>
        </h:selectOneMenu>
        <h:outputLabel value="Minor "/>
        <h:selectManyListbox id="minorSelectManyListbox">
          <f:selectItem itemValue="Computer Science"/>
          <f:selectItem itemValue="Mathematics"/>
          <f:selectItem itemValue="English"/>
        </h:selectManyListbox>
      </h:panelGrid>

      <!-- Use check boxes -->
      <h:panelGrid columns="4">
        <h:outputLabel value="Hobby: "/>
        <h:selectManyCheckbox id="hobbySelectManyCheckbox">
          <f:selectItem itemValue="Tennis"/>
          <f:selectItem itemValue="Golf"/>
          <f:selectItem itemValue="Ping Pong"/>
        </h:selectManyCheckbox>
      </h:panelGrid>

      <!-- Use text area -->
      <h:panelGrid columns="1">
        <h:outputLabel>Remarks:</h:outputLabel>
        <h:inputTextarea id="remarksInputTextarea"
                         style="width:400px; height:50px;" />
      </h:panelGrid>

      <!-- Use command button -->
      <h:commandButton value="Register" />
    </h:form>
  </h:body>
</html>
```

The tags with prefix `f` are in the JSF core tag library. Line 6

jsf core xmlns

```
xmlns:f="http://xmlns.jcp.org/jsf/core">
```

locates the library for these tags.

h:graphicImage

The `h:graphicImage` tag displays an image in the file usIcon.gif (line 14). The file is located in the /resources/image folder. In JSF 2.0, all resources (image files, audio files, and CCS files) should be placed under the `resources` folder under the **Web Pages** node. You can create these folders as follows:

Step 1: Right-click the Web Pages node in the project pane to display a context menu and choose *New*, *Folder* to display the New Folder dialog box. (If *Folder* is not in the context menu, choose *Other* to locate it.)

Step 2: Enter `resources` as the Folder Name and click *Finish* to create the `resources` folder, as shown in Figure 39.12.

Step 3: Right-click the `resources` node in the project pane to create the image folder under resources. You can now place usIcon.gif under the image folder.

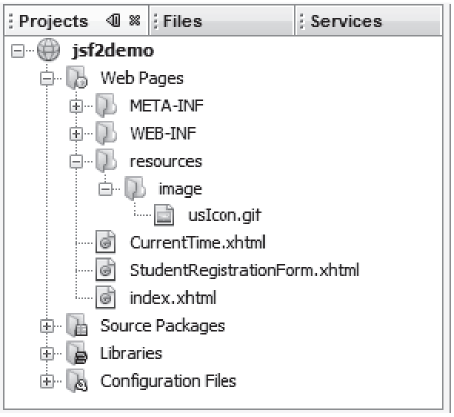


FIGURE 39.12 The resources folder was created.

h:panelGrid

JSF provides `h:panelGrid` and `h:panelGroup` elements to contain and layout subelements. `h:panelGrid` places the elements in a grid like the JavaFX `GridPane`. `h:panelGrid` places the elements in a grid with the specified number of columns. Lines 18–25 place six elements (labels and input texts) that are in an `h:panelGrid`. The `columns` attribute specifies that each row in the grid has `6` columns. The elements are placed into a row from left to right in the order they appear in the facelet. When a row is full, a new row is created to hold the elements. We used `h:panelGrid` in this example. You may replace it with `h:panelGroup` to see how the elements would be arranged.

the `style` attribute

You may use the `style` attribute with a JSF html tag to specify the CSS style for the element and its subelements. The `style` attribute in line 18 specifies color green for all elements in this `h:panelGrid` element.

h:outputLabel

The `h:outputLabel` element is for displaying a label (line 19). The `value` attribute specifies the label's text.

h:inputText

The `h:inputText` element is for displaying a text input box for the user to enter a text (line 20). The `id` attribute is useful for other elements or the server program to reference this element.

h:selectOneRadio

The `h:selectOneRadio` element is for displaying a group of radio buttons (line 30). Each radio button is defined using an `f:selectItem` element (lines 31–34).

`h:selectOneMenu`

The `h:selectOneMenu` element is for displaying a combo box (line 41). Each item in the combo box is defined using an `f:selectItem` element (lines 42 and 43).

h:selectManyListbox

The `h:selectManyListbox` element is for displaying a list for the user to choose multiple items in a list (line 46). Each item in the list is defined using an `f:selectItem` element (lines 47–49).

h:selectManyCheckbox

The `h:selectManyCheckbox` element is for displaying a group of check boxes (line 56). Each item in the check box is defined using an `f:selectItem` element (lines 57–59).

`h:selectTextarea`

The `h:selectTextarea` element is for displaying a text area for multiple lines of input (line 66). The `style` attribute is used to specify the width and height of the text area (line 67).

`h:commandButton`

The `h:commandButton` element is for displaying a button (line 71). When the button is clicked, an action is performed. The default action is to request the same page from the server. The next section shows how to process the form.

Check Point

39.3.1 What is the name space for JSF tags with prefix `h` and prefix `f`?

39.3.2 Describe the use of the following tags?

`h:form`, `h:panelGroup`, `h:panelGrid`, `h:inputText`, `h:outputText`, `h:inputTextArea`, `h:inputSecret`, `h:outputLabel`, `h:outputLink`, `h:selectOneMenu`, `h:selectOneRadio`, `h:selectBooleanCheckbox`, `h:selectOneListbox`, `h:selectManyListbox`, `h:selectItem`, `h:message`, `h:dataTable`, `h:columm`, `h:graphicImage`

39.4 Processing the Form

Key Point

Processing forms is a common task for Web programming. JSF provides tools for processing forms.

The preceding section introduced how to display a form using common JSF elements. This section shows how to obtain and process the input.

To obtain input from the form, simply bind each input element with a property in a managed bean. We now define a managed bean named `registration` as shown in Listing 39.5.

Listing 39.5 `RegistrationJSFBean.java`

managed bean
request scope
property `lastName`

```java
package jsf2demo;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named(value = "registration")
@RequestScoped
public class RegistrationJSFBean {
  private String lastName;
  private String firstName;
  private String mi;
  private String gender;
  private String major;
  private String[] minor;
  private String[] hobby;
  private String remarks;

  public RegistrationJSFBean() {
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getMi() {
    return mi;
  }

  public void setMi(String mi) {
    this.mi = mi;
  }

  public String getGender() {
    return gender;
  }

  public void setGender(String gender) {
    this.gender = gender;
  }

  public String getMajor() {
    return major;
  }

  public void setMajor(String major) {
    this.major = major;
  }

  public String[] getMinor() {
    return minor;
  }

  public void setMinor(String[] minor) {
    this.minor = minor;
  }

  public String[] getHobby() {
    return hobby;
  }

  public void setHobby(String[] hobby) {
    this.hobby = hobby;
  }

  public String getRemarks() {
    return remarks;
  }

  public void setRemarks(String remarks) {
    this.remarks = remarks;
  }
```

getResponse()

```

  public String getResponse() {
    if (lastName == null)
      return ""; // Request has not been made
    else {
      String allMinor = "";
      for (String s: minor) {
        allMinor += s + " ";
      }

      String allHobby = "";
      for (String s: hobby) {
        allHobby += s + " ";
      }

      return "<p style=\"color:red\">You entered <br />" +
        "Last Name: " + lastName + "<br />" +
        "First Name: " + firstName + "<br />" +
        "MI: " + mi + "<br />" +
        "Gender: " + gender + "<br />" +
        "Major: " + major + "<br />" +
        "Minor: " + allMinor + "<br />" +
        "Hobby: " + allHobby + "<br />" +
        "Remarks: " + remarks + "</p>";
    }
  }
}
```

bean properties

The **`RegistrationJSFBean`** class is a managed bean that defines the properties **`lastName`**, **`firstName`**, **`mi`**, **`gender`**, **`major`**, **`minor`**, and **`remarks`**, which will be bound to the elements in the JSF registration form.

The registration form can now be revised as shown in Listing 39.6. Figure 39.13 shows that new JSF page displays the user input upon clicking the *Register* button.

Listing 39.6 ProcessStudentRegistrationForm.xhtml

jsf core namespace

bind `lastName`

bind `firstName`

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
  <h:head>
    <title>Student Registration Form</title>
  </h:head>
  <h:body>
    <h:form>
      <!-- Use h:graphicImage -->
      <h3>Student Registration Form
        <h:graphicImage name="usIcon.gif" library="image"/>
      </h3>

      <!-- Use h:panelGrid -->
      <h:panelGrid columns="6" style="color:green">
        <h:outputLabel value="Last Name"/>
        <h:inputText id="lastNameInputText"
                     value="#{registration.lastName}"/>
        <h:outputLabel value="First Name" />
        <h:inputText id="firstNameInputText"
                     value="#{registration.firstName}"/>
        <h:outputLabel value="MI" />
        <h:inputText id="miInputText" size="1"
                     value="#{registration.mi}"/>
        </h:panelGrid>

        <!-- Use radio buttons -->
        <h:panelGrid columns="2">
          <h:outputLabel>Gender </h:outputLabel>
          <h:selectOneRadio id="genderSelectOneRadio"
                            value="#{registration.gender}">
          <f:selectItem itemValue="Male"
                        itemLabel="Male"/>
          <f:selectItem itemValue="Female"
                        itemLabel="Female"/>
        </h:selectOneRadio>
      </h:panelGrid>

      <!-- Use combo box and list -->
      <h:panelGrid columns="4">
        <h:outputLabel value="Major "/>
        <h:selectOneMenu id="majorSelectOneMenu"
                         value="#{registration.major}">
          <f:selectItem itemValue="Computer Science"/>
          <f:selectItem itemValue="Mathematics"/>
        </h:selectOneMenu>
        <h:outputLabel value="Minor "/>
        <h:selectManyListbox id="minorSelectManyListbox"
                             value="#{registration.minor}">
          <f:selectItem itemValue="Computer Science"/>
          <f:selectItem itemValue="Mathematics"/>
          <f:selectItem itemValue="English"/>
        </h:selectManyListbox>
      </h:panelGrid>

      <!-- Use check boxes -->
      <h:panelGrid columns="4">
        <h:outputLabel value="Hobby: "/>
        <h:selectManyCheckbox id="hobbySelectManyCheckbox"
                              value="#{registration.hobby}">
          <f:selectItem itemValue="Tennis"/>
          <f:selectItem itemValue="Golf"/>
          <f:selectItem itemValue="Ping Pong"/>
        </h:selectManyCheckbox>
      </h:panelGrid>

      <!-- Use text area -->
      <h:panelGrid columns="1">
        <h:outputLabel>Remarks:</h:outputLabel>
        <h:inputTextarea id="remarksInputTextarea"
                         style="width:400px; height:50px;"
                         value="#{registration.remarks}"/>
      </h:panelGrid>

      <!-- Use command button -->
      <h:commandButton value="Register" />
      <br />
      <h:outputText escape="false" style="color:red"
                    value="#{registration.response}" />
    </h:form>
  </h:body>
</html>
```

bind mi

bind gender

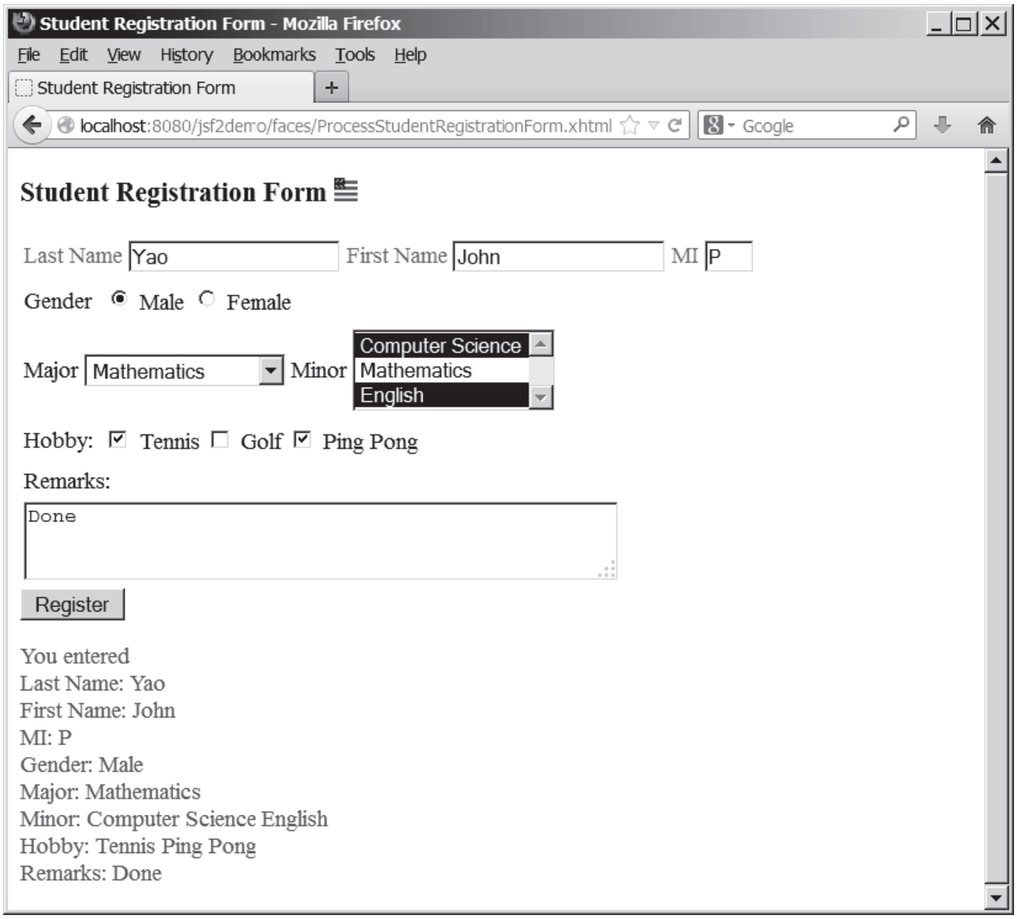
bind major

bind minor

bind hobby

bind remarks

bind response


Student Registration Form - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Student Registration Form
localhost:8080/jsf2demo/faces/ProcessStudentRegistrationForm.xhtml
Google
Register

Student Registration Form

Last Name Yao First Name John MI P

Gender ● Male ○ Female

Major Mathematics Minor Computer Science, Mathematics, English

Hobby: ☑ Tennis ☐ Golf ☑ Ping Pong

Remarks:

Done

You entered
Last Name: Yao
First Name: John
MI: P
Gender: Male
Major: Mathematics
Minor: Computer Science English
Hobby: Tennis Ping Pong
Remarks: Done

FIGURE 39.13 The user input is collected and displayed after clicking the Register button.

binding input texts

The new JSF form in this listing binds the `h:inputText` element for last name, first name, and mi with the properties `lastName`, `firstName`, and `mi` in the managed bean (lines 21, 24, and 27). When the *Register* button is clicked, the page is sent to the server, which invokes the setter methods to set the properties in the managed bean.

binding radio buttons

The `h:selectOneRadio` element is bound to the `gender` property (line 34). Each radio button has an `itemValue`. The selected radio button's `itemValue` is set to the `gender` property in the bean when the page is sent to the server.

binding combo box

The `h:selectOneMenu` element is bound to the `major` property (line 46). When the page is sent to the server, the selected item is returned as a string and is set to the `major` property.

binding list box

The `h:selectManyListbox` element is bound to the `minor` property (line 52). When the page is sent to the server, the selected items are returned as an array of strings and set to the `minor` property.

binding check boxes

The `h:selectManyCheckbox` element is bound to the `hobby` property (line 63). When the page is sent to the server, the checked boxes are returned as an array of `itemValues` and set to the `hobby` property.

binding text area

The `h:selectTextarea` element is bound to the `remarks` property (line 75). When the page is sent to the server, the content in the text area is returned as a string and set to the `remarks` property.

binding response

The `h:outputText` element is bound to the `response` property (line 82). This is a read-only property in the bean. It is `""` if `lastName` is `null` (lines 86 and 87 in Listing 39.5). When the page is returned to the client, the `response` property value is displayed in the output text element (line 82).

The `h:outputText` element's `escape` attribute is set to `false` (line 81) to enable the contents to be displayed in HTML. By default, the `escape` attribute is `true`, which indicates the contents are considered regular text.

escape attribute

39.4.1 In the `h:outputText` tag, what is the `escape` attribute for?

39.4.2 Does every GUI component tag in JSF have the style attribute?

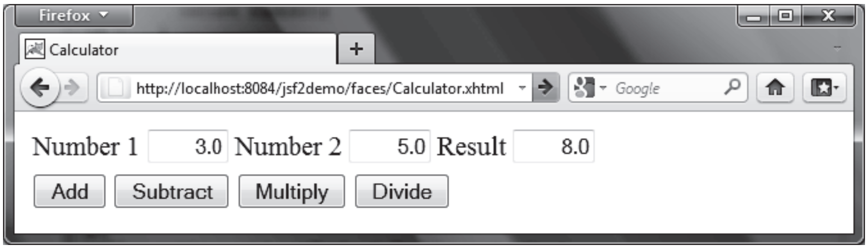
Check Point

39.5 Case Study: Calculator

This section gives a case study on using GUI elements and processing forms.

Key Point

This section uses JSF to develop a calculator to perform addition, subtraction, multiplication, and division, as shown in Figure 39.14.



Number 1 3.0 Number 2 5.0 Result 8.0

FIGURE 39.14 This JSF application enables you to perform addition, subtraction, multiplication, and division.

Here are the steps to develop this project:

Step 1. Create a new managed bean named `calculator` with the request scope as shown in Listing 39.7, CalculatorJSFBean.java.

create managed bean

Step 2. Create a JSF facelet in Listing 39.8, Calculator.xhtml.

create JSF facelet

LISTING 39.7 `CalculatorJSFBean.java`

```
package jsf2demo;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

@Named(value = "calculator")
@RequestScoped
public class CalculatorJSFBean {
  private Double number1;
  private Double number2;
  private Double result;

  public CalculatorJSFBean() {
  }

  public Double getNumber1() {
    return number1;
  }

  public Double getNumber2() {
    return number2;
  }

```

property `number1`
property `number2`
property `result`

```
                 24    public Double getResult() {
                 25      return result;
                 26    }
                 27
                 28    public void setNumber1(Double number1) {
                 29      this.number1 = number1;
                 30    }
                 31
                 32    public void setNumber2(Double number2) {
                 33      this.number2 = number2;
                 34    }
                 35
                 36    public void setResult(Double result) {
                 37      this.result = result;
                 38    }
                 39
add              40    public void add() {
                 41      result = number1 + number2;
                 42    }
                 43
subtract         44    public void subtract() {
                 45      result = number1 - number2;
                 46    }
                 47
divide           48    public void divide() {
                 49      result = number1 / number2;
                 50    }
                 51
multiply         52    public void multiply() {
                 53      result = number1 * number2;
                 54    }
                 55  }
```

The managed bean has three properties `number1`, `number2`, and `result` (lines 9–38). The methods `add()`, `subtract()`, `divide()`, and `multiply()` add, subtract, multiply, and divide `number1` with `number2` and assigns the result to `result` (lines 40–54).

LISTING 39.8 `Calculator.xhtml`

```
                 <?xml version='1.0' encoding='UTF-8' ?>
                 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
                 <html xmlns="http://www.w3.org/1999/xhtml"
                       xmlns:h="http://xmlns.jcp.org/jsf/html">
                   <h:head>
                     <title>Calculator</title>
                   </h:head>
                   <h:body>
                     <h:form>
                       <h:panelGrid columns="6">
                         <h:outputLabel value="Number 1"/>
                         <h:inputText id="number1InputText" size ="4"
right align                           style="text-align: right"
bind text input                       value="#{calculator.number1}"/>
                         <h:outputLabel value="Number 2" />
                         <h:inputText id="number2InputText" size ="4"
                                      style="text-align: right"
                                      value="#{calculator.number2}"/>
                         <h:outputLabel value="Result" />
                         <h:inputText id="resultInputText" size ="4"
```

```
22                       style="text-align: right"
23                       value="#{calculator.result}"/>
24        </h:panelGrid>
25
26        <h:panelGrid columns="4">
27          <h:commandButton value="Add"
28                           action ="#{calculator.add}"/>
29          <h:commandButton value="Subtract"
30                           action ="#{calculator.subtract}"/>
31          <h:commandButton value="Multiply"
32                           action ="#{calculator.multiply}"/>
33          <h:commandButton value="Divide"
34                           action ="#{calculator.divide}"/>
35        </h:panelGrid>
36      </h:form>
37    </h:body>
38  </html>
```

action

Three text input components along with their labels are placed in the grid panel (lines 11–24). Four button components are placed in the grid panel (lines 26–35).

The bean property `number1` is bound to the text input for Number 1 (line 15). The CSS style `text-align: right` (line 14) specifies that the text is right aligned in the input box.

The `action` attribute for the *Add* button is set to the `add` method in the calculator bean (line 28). When the *Add* button is clicked, the `add` method in the bean is invoked to add `number1` with `number2` and assign the result to `result`. Since the `result` property is bound to the Result input text (line 23), the new result is now displayed in the text input field.

39.6 Session Tracking

You can create a managed bean at the application scope, session scope, view scope, or request scope.

Key Point

JSF supports session tracking for managed beans at the application scope, session scope, view scope, and request scope. The *scope* is the lifetime of a bean. A *request*-scoped bean is alive in a single HTTP request. After the request is processed, the bean is no longer alive. A *view*-scoped bean lives as long as you are in the same JSF page. A *session*-scoped bean is alive for the entire Web session between a client and the server. An *application*-scoped bean lives as long as the Web application runs. In essence, a request-scoped bean is created once for a request; a view-scoped bean is created once for the view; a session-scoped bean is created once for the entire session; and an application-scoped bean is created once for the entire application. A managed bean with a session scope must be serializable because the system may need to free resources during and session and stores the bean to a file if the bean is not used for a while. When the bean is used again, the system will restore the bean to the memory.

scope
request scope
view scope
session scope
application scope

Consider the following example that prompts the user to guess a number. When the page starts, the program randomly generates a number between `0` and `99`. This number is stored in a bean. When the user enters a guess, the program checks the guess with the random number in the bean and tells the user whether the guess is too high, too low, or just right, as shown in Figure 39.15.

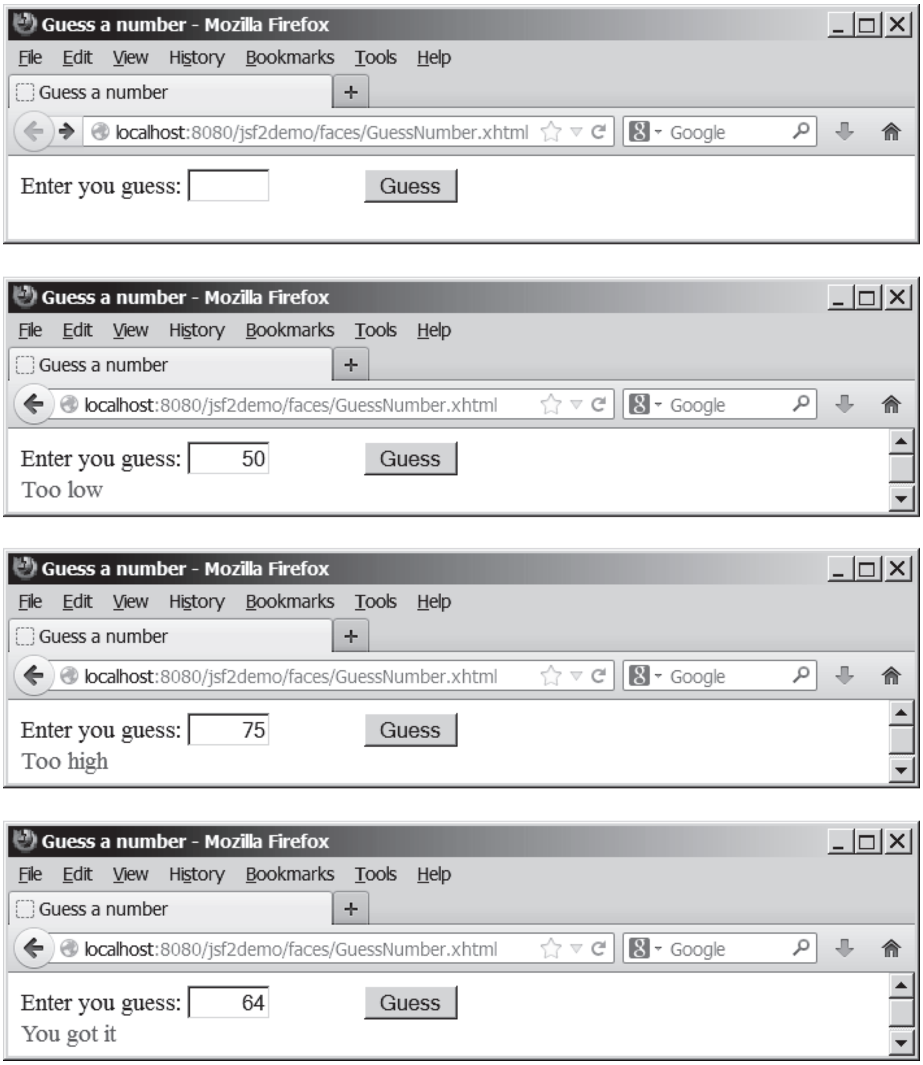
Here are the steps to develop this project:

Step 1. Create a new managed bean named `guessNumber` with the view scope as shown in Listing 39.9, GuessNumberJSFBean.java.

create managed bean

Step 2. Create a JSF facelet in Listing 39.10, GuessNumber.xhtml.

create JSF facelet


Guess a number - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Guess a number
localhost:8080/jsf2demo/faces/GuessNumber.xhtml
Google
Guess

Enter you guess:

Enter you guess: 50
Too low

Enter you guess: 75
Too high

Enter you guess: 64
You got it

FIGURE 39.15 The user enters a guess and the program displays the result.

LISTING 39.9 GuessNumberJSFBean.java

view scope

random number

guess by user

create random number

getter method

```java
package jsf2demo;

import javax.inject.Named;
import javax.faces.view.ViewScoped;

@Named(value = "guessNumber")
@ViewScoped
public class GuessNumberJSFBean {
  private int number;
  private String guessString;

  public GuessNumberJSFBean() {
   number = (int)(Math.random() * 100);
  }

  public String getGuessString() {
    return guessString;
  }
19
20    public void setGuessString(String guessString) {
21      this.guessString = guessString;
22    }
23
24    public String getResponse() {
25      if (guessString == null)
26        return ""; // No user input yet
27
28      int guess = Integer.parseInt(guessString);
29      if (guess < number)
30        return "Too low";
31      else if (guess == number)
32        return "You got it";
33      else
34        return "Too high";
35    }
36  }
```

setter method

get response

check guess

The managed bean uses the `@ViewScope` annotation (line 7) to set up the view scope for the bean. The view scope is most appropriate for this project. The bean is alive as long as the view is not changed. The bean is created when the page is displayed for the first time. A random number between `0` and `99` is assigned to `number` (line 13) when the bean is created. This number will not change as long as the bean is alive in the same view.

The `getResponse` method converts `guessString` from the user input to an integer (line 28) and determines if the guess is too low (line 30), too high (line 34), and just right (line 32).

LISTING 39.10 GuessNumber.xhtml

```
 1  <?xml version='1.0' encoding='UTF-8' ?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 4  <html xmlns="http://www.w3.org/1999/xhtml"
 5        xmlns:h="http://xmlns.jcp.org/jsf/html">
 6    <h:head>
 7      <title>Guess a number</title>
 8    </h:head>
 9    <h:body>
10      <h:form>
11        <h:outputLabel value="Enter you guess: "/>
12        <h:inputText style="text-align: right; width: 50px"
13                     id="guessInputText"
14                     value="#{guessNumber.guessString}"/>
15        <h:commandButton style="margin-left: 60px" value="Guess" />
16        <br />
17        <h:outputText style="color: red"
18                      value="#{guessNumber.response}" />
19      </h:form>
20    </h:body>
21  </html>
```

bind text input

bind text output

The bean property `guessString` is bound to the text input (line 14). The CSS style `text-align: right` (line 13) specifies that the text is right aligned in the input box.

The CSS style `margin-left: 60px` (line 15) specifies that the command button has a left margin of 60 pixels.

The bean property `response` is bound to the text output (line 18). The CSS style `color: red` (line 17) specifies that the text is displayed in red in the output box.

scope

The project uses the **view** scope. What happens if the scope is changed to the request scope? Every time the page is refreshed, JSF creates a new bean with a new random number. What happens if the scope is changed to the **session** scope? The bean will be alive as long as the browser is alive. What happens if the scope is changed to the **application** scope? The bean will be created once when the application is launched from the server. So every client will use the same random number.

Check Point

39.6.1 What is a scope? What are the available scopes in JSF? Explain request scope, view scope, session scope, and application scope. How do you set a request scope, view scope, session scope, and application scope in a managed bean?

39.6.2 What happens if the bean scope in Listing 39.9 GuessNumberJSFBean.java is changed to request?

39.6.3 What happens if the bean scope in Listing 39.9 GuessNumberJSFBean.java is changed to session?

39.6.4 What happens if the bean scope in Listing 39.9 GuessNumberJSFBean.java is changed to application?

39.7 Validating Input

Key Point

JSF provides tools for validating user input.

In the preceding `GuessNumber` page, an error would occur if you entered a noninteger in the input box before clicking the *Guess* button. One way to fix the problem is to check the text field before processing any event. But a better way is to user the validators. You can use the standard validator tags in the JSF Core Tag Library or create custom validators. Table 39.2 lists some JSF input validator tags.

TABLE 39.2 JSF Input Validator Tags

JSF Tag	*Description*
`f:validateLength`	validates the length of the input.
`f:validateDoubleRange`	validates whether numeric input falls within acceptable range of double values.
`f:validateLongRange`	validates whether numeric input falls within acceptable range of long values.
`f:validateRequired`	validates whether a field is not empty.
`f:validateRegex`	validates whether the input matches a regualar expression.
`f:validateBean`	invokes a custom method in a bean to perform custom validation.

Consider the following example that displays a form for collecting user input as shown in Figure 39.16. All text fields in the form must be filled. If not, error messages are displayed. The SSN must be formatted correctly. If not, an error is displayed. If all input are correct, clicking *Submit* displays the result in an output text, as shown in Figure 39.17.

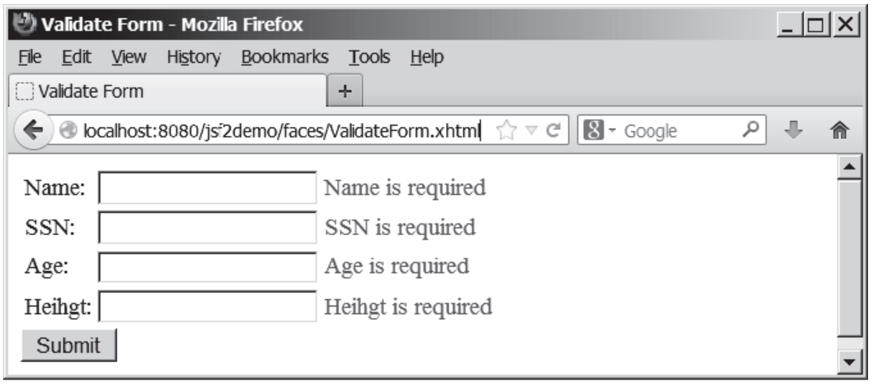
Here are the steps to create this project.

Step 1. Create a new page in Listing 39.11, ValidateForm.xhtml.

Step 2. Create a new managed bean named `validateForm`, as shown in Listing 39.12.

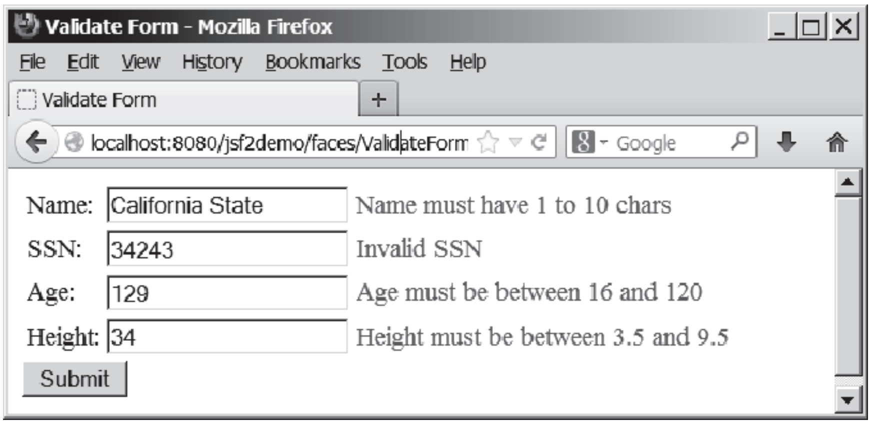
LISTING 39.11 ValidateForm.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
  <h:head>
```


Validate Form - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Validate Form
localhost:8080/jsf2demo/faces/ValidateForm.xhtml
Google
Submit

Name: Name is required
SSN: SSN is required
Age: Age is required
Heihgt: Heihgt is required

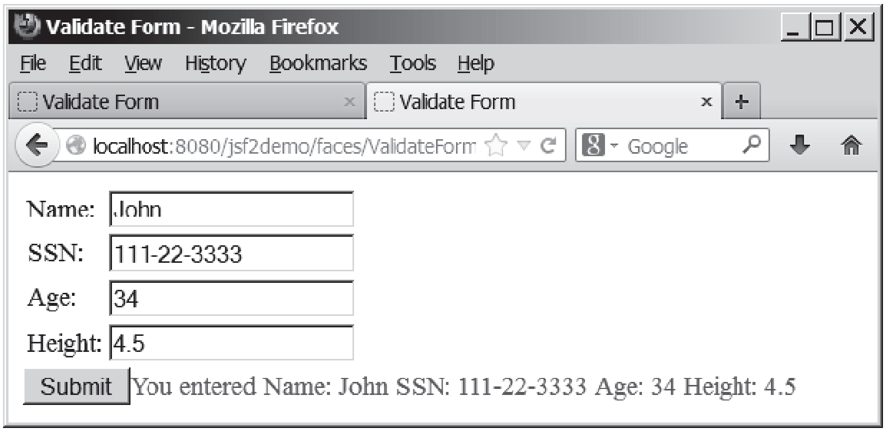
(a) *The required messages are displayed if input is required, but empty.*


Validate Form - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Validate Form
localhost:8080/jsf2demo/faces/ValidateForm
Google
Submit

Name: California State Name must have 1 to 10 chars
SSN: 34243 Invalid SSN
Age: 129 Age must be between 16 and 120
Height: 34 Height must be between 3.5 and 9.5

(b) *Error messages are displayed if input is incorrect.*

Figure 39.16 The input fields are validated.


Validate Form - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Validate Form
Validate Form
localhost:8080/jsf2demo/faces/ValidateForm
Google
Submit

Name: John
SSN: 111-22-3333
Age: 34
Height: 4.5
You entered Name: John SSN: 111-22-3333 Age: 34 Height: 4.5

Figure 39.17 The correct input values are displayed.

```
    <title>Validate Form</title>
  </h:head>
  <h:body>
    <h:form>
      <h:panelGrid columns="3">
        <h:outputLabel value="Name:"/>
```

required input
required message
validator message
validate length
message element
validate regex
validate integer range
validate double range

```
14          <h:inputText id="nameInputText" required="true"
15            requiredMessage="Name is required"
16            validatorMessage="Name must have 1 to 10 chars"
17            value="#{validateForm.name}">
18            <f:validateLength minimum="1" maximum="10" />
19          </h:inputText>
20          <h:message for="nameInputText" style="color:red"/>
21
22          <h:outputLabel value="SSN:" />
23          <h:inputText id="ssnInputText" required="true"
24            requiredMessage="SSN is required"
25            validatorMessage="Invalid SSN"
26            value="#{validateForm.ssn}">
27            <f:validateRegex pattern="[\d]{3}-[\d]{2}-[\d]{4}"/>
28          </h:inputText>
29          <h:message for="ssnInputText" style="color:red"/>
30
31          <h:outputLabel value="Age:" />
32          <h:inputText id="ageInputText" required="true"
33            requiredMessage="Age is required"
34            validatorMessage="Age must be between 16 and 120"
35            value="#{validateForm.ageString}">
36            <f:validateLongRange minimum="16" maximum="120"/>
37          </h:inputText>
38          <h:message for="ageInputText" style="color:red"/>
39
40          <h:outputLabel value="Height:" />
41          <h:inputText id="heightInputText" required="true"
42            requiredMessage="Height is required"
43            validatorMessage="Height must be between 3.5 and 9.5"
44            value="#{validateForm.heightString}">
45            <f:validateDoubleRange minimum="3.5" maximum="9.5"/>
46          </h:inputText>
47          <h:message for="heightInputText" style="color:red"/>
48        </h:panelGrid>
49
50        <h:commandButton value="Submit" />
51
52        <h:outputText style="color:red"
53                      value="#{validateForm.response}" />
54      </h:form>
55    </h:body>
56  </html>
```

`required` attribute
`requiredMessage`

For each input text field, set its `required` attribute `true` (lines 14, 23, 32, and 41) to indicate that an input value is required for the field. When a required input field is empty, the `requiredMessage` is displayed (lines 15, 24, 33, and 42).

`validatorMessage`
`f:validateLength`

The `validatorMessage` attribute specifies a message to be displayed if the input field is invalid (line 16). The `f:validateLength` tag specifies the minimum or maximum length of the input (line 18). JSF will determine whether the input length is valid.

`h:message`

The `h:message` element displays the `validatorMessage` if the input is invalid. The element's `for` attribute specifies the `id` of the element for which the message will be displayed (line 20).

`f:validateRegex`

The `f:validateRegex` tag specifies a regular expression for validating the input (line 27). For information on regular expression, see Appendix H.

f:validateLongRange

The `f:validateLongRange` tag specifies a range for an integer input using the `minimum` and `maximum` attributes (line 45). In this project, a valid age value is between `16` and `120`.

f:validateDoubleRange

The `f:validateDoubleRange` tag specifies a range for a double input using the `minimum` and `maximum` attributes (line 36). In this project, a valid height value is between `3.5` and `9.5`.

LISTING 39.12 `ValidateFormJSFBean.java`

```java
package jsf2demo;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named(value = "validateForm")
@RequestScoped
public class ValidateFormJSFBean {
  private String name;
  private String ssn;
  private String ageString;
  private String heightString;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getSsn() {
    return ssn;
  }

  public void setSsn(String ssn) {
    this.ssn = ssn;
  }

  public String getAgeString() {
    return ageString;
  }

  public void setAgeString(String ageString) {
    this.ageString = ageString;
  }

  public String getHeightString() {
    return heightString;
  }

  public void setHeightString(String heightString) {
    this.heightString = heightString;
  }

  public String getResponse() {
    if (name == null || ssn == null || ageString == null
            || heightString == null) {
      return "";
    }
    else {
      return "You entered " +
        " Name: " + name +
        " SSN: " + ssn +
        " Age: " + ageString +
        " Height: " + heightString;
    }
  }
}
```

some input not set

If an input is invalid, its value is not set to the bean. So only when all input are correct, the `getResponse()` method will return all input values (lines 46–58).

Check Point

39.7.1 Write a tag that validates an input text with minimal length of `2` and maximum `12`.

39.7.2 Write a tag that validates an input text for SSN using a regular expression.

39.7.3 Write a tag that validates an input text for a double value with minimal `4.5` and maximum `19.9`.

39.7.4 Write a tag that validates an input text for an integer value with minimal `4` and maximum `20`.

39.7.5 Write a tag that makes an input text required.

39.8 Binding Database with Facelets

Key Point

You can bind a database in JSF applications.

Often you need to access a database from a Web page. This section gives examples of building Web applications using databases.

Consider the following example that lets the user choose a course, as shown in Figure 39.18. After a course is selected in the combo box, the students enrolled in the course are displayed in the table, as shown in Figure 39.19. In this example, all the course titles in the `Course` table are bound to the combo box and the query result for the students enrolled in the course is bound to the table.

Here are the steps to create this project:

Managed bean

Step 1. Create a managed bean named `courseName` with application scope, as shown in Listing 39.13.

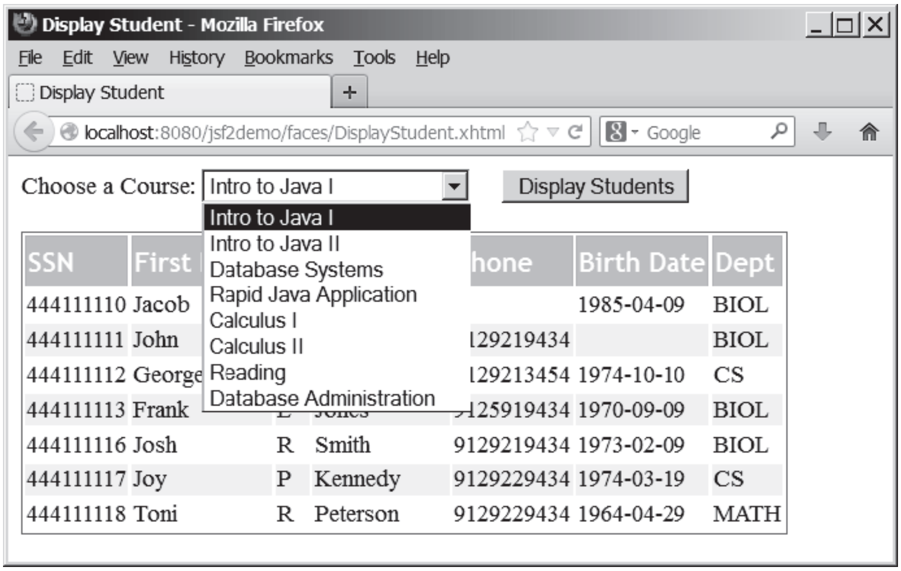
JSF page

Step 2. Create a JSF in Listing 39.14, DisplayStudent.xhtml.

style sheet

Step 3. Create a cascading style sheet for formatting the table as follows:

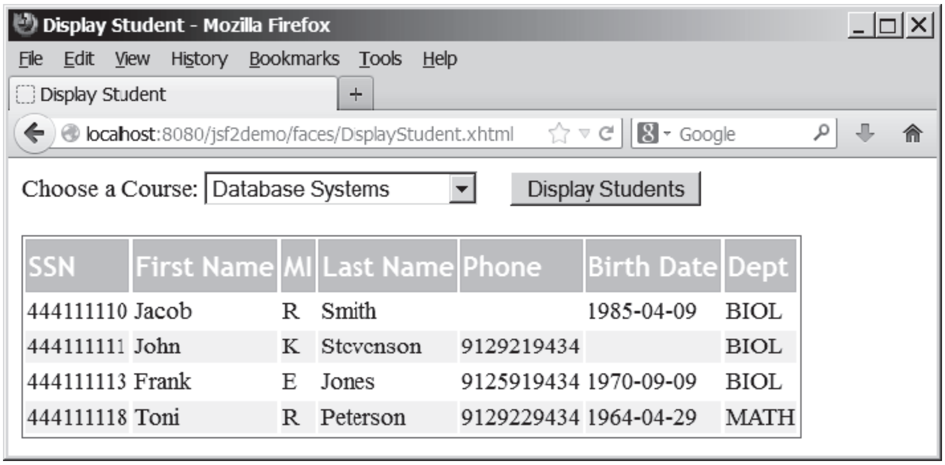
Step 3.1. Right-click the `resources` node to choose *New*, *Others* to display the New File dialog box, as shown in Figure 39.20.


Display Student - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Display Student
+
localhost:8080/jsf2demo/faces/DisplayStudent.xhtml
Google
Display Students
Intro to Java I
Intro to Java II
Database Systems
Rapid Java Application
Calculus I
Calculus II
Reading
Database Administration

Choose a Course: Intro to Java I

SSN	First			hone	Birth Date	Dept
444111110	Jacob				1985-04-09	BIOL
444111111	John			129219434		BIOL
444111112	George			129213454	1974-10-10	CS
444111113	Frank			125919434	1970-09-09	BIOL
444111116	Josh	R	Smith	9129219434	1973-02-09	BIOL
444111117	Joy	P	Kennedy	9129229434	1974-03-19	CS
444111118	Toni	R	Peterson	9129229434	1964-04-29	MATH

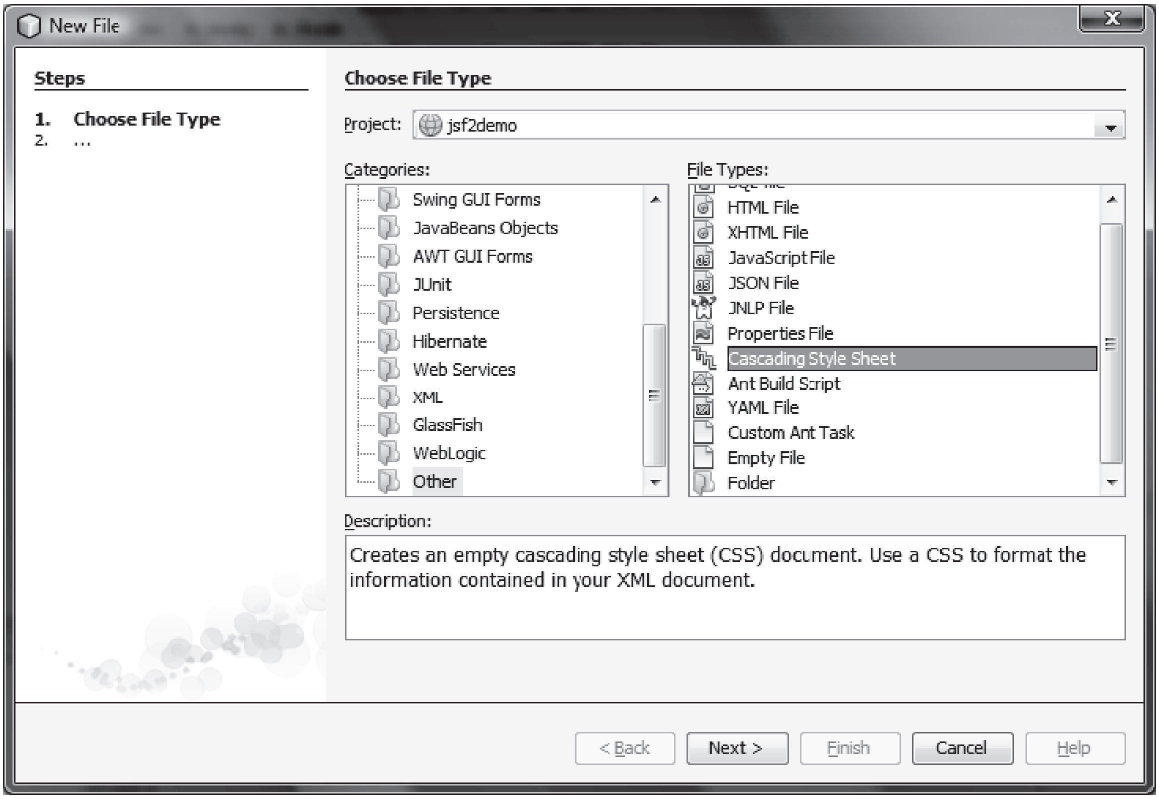
FIGURE 39.18 You need to choose a course and display the students enrolled in the course.


Display Student - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Display Student
localhost:8080/jsf2demo/faces/DisplayStudent.xhtml
Google
Display Students

Choose a Course: Database Systems

SSN	First Name	MI	Last Name	Phone	Birth Date	Dept
444111110	Jacob	R	Smith		1985-04-09	BIOL
444111111	John	K	Stevenson	9129219434		BIOL
444111113	Frank	E	Jones	9125919434	1970-09-09	BIOL
444111118	Toni	R	Peterson	9129229434	1964-04-29	MATH

FIGURE 39.19 The table displays the students enrolled in the course.


New File
Steps
1. Choose File Type
2. ...
Choose File Type
Categories:
Swing GUI Forms
JavaBeans Objects
AWT GUI Forms
JUnit
Persistence
Hibernate
Web Services
XML
GlassFish
WebLogic
Other
File Types:
HTML File
XHTML File
JavaScript File
JSON File
JNLP File
Properties File
Cascading Style Sheet
Ant Build Script
YAML File
Custom Ant Task
Empty File
Folder
< Back
Next >
Finish
Cancel
Help

Project: jsf2demo

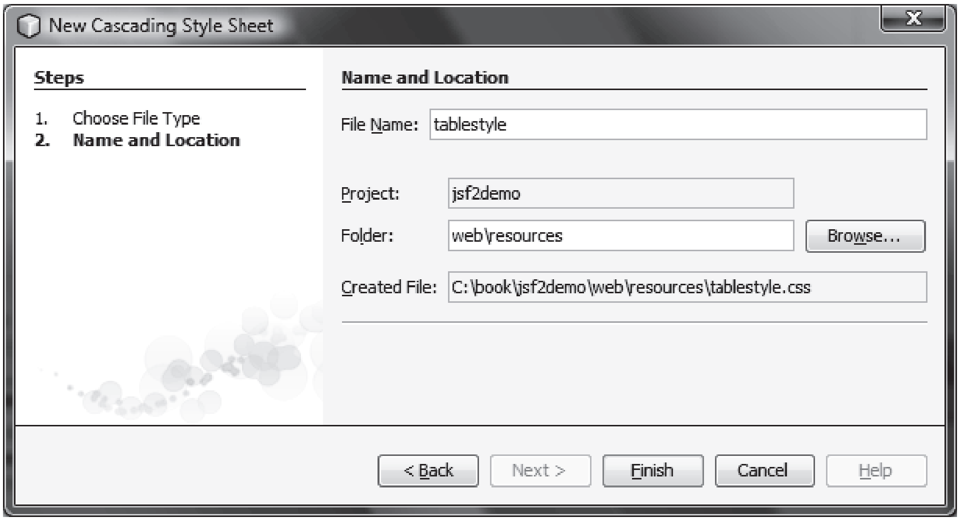
Description:
Creates an empty cascading style sheet (CSS) document. Use a CSS to format the information contained in your XML document.

FIGURE 39.20 You can create CSS files for Web project in NetBenas.

Step 3.2. Choose `Other` in the Categories section and `Cascading Style Sheet` in the File Types section to display the New Cascading Style Sheet dialog box, as shown in Figure 39.21.

Step 3.3. Enter `tablestyle` as the File Name and click *Finish* to create tablestyle.css under the resources node.

Step 3.4. Define the CSS style as shown in Listing 39.15.



File Name: tablestyle

Project: jsf2demo

Folder: web\resources

Created File: C:\book\jsf2demo\web\resources\tablestyle.css

FIGURE 39.21 The New Cascading Style Sheet dialog box creates a new style sheet file.

LISTING 39.13 CourseNameJSFBean.java

application scope

initialize JDBC

connect to database

get course titles

execute SQL

```java
package jsf2demo;

import java.sql.*;
import java.util.ArrayList;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Named;

@Named(value = "courseName")
@ApplicationScoped
public class CourseNameJSFBean {
  private PreparedStatement studentStatement = null;
  private String choice; // Selected course
  private String[] titles; // Course titles

  /** Creates a new instance of CourseName */
  public CourseNameJSFBean() {
    initializeJdbc();
  }

  /** Initialize database connection */
  private void initializeJdbc() {
    try {
      Class.forName("com.mysql.jdbc.Driver");
      System.out.println("Driver loaded");

      // Connect to the sample database
      Connection connection = DriverManager.getConnection(
        "jdbc:mysql://localhost/javabook", "scott", "tiger");

      // Get course titles
      PreparedStatement statement = connection.prepareStatement(
        "select title from course");

      ResultSet resultSet = statement.executeQuery();

      // Store resultSet into array titles
      ArrayList<String> list = new ArrayList<>();
38        while (resultSet.next()) {
39          list.add(resultSet.getString(1));
40        }
41        titles = new String[list.size()]; // Array for titles
42        list.toArray(titles); // Copy strings from list to array
43
44        // Define a SQL statement for getting students
45        studentStatement = connection.prepareStatement(
46          "select Student.ssn, "
47          + "student.firstName, Student.mi, Student.lastName, "
48          + "Student.phone, Student.birthDate, Student.street, "
49          + "Student.zipCode, Student.deptId "
50          + "from Student, Enrollment, Course "
51          + "where Course.title = ? "
52          + "and Student.ssn = Enrollment.ssn "
53          + "and Enrollment.courseId = Course.courseId;");
54      }
55      catch (Exception ex) {
56        ex.printStackTrace();
57      }
58    }
59
60    public String[] getTitles() {
61      return titles;
62    }
63
64    public String getChoice() {
65      return choice;
66    }
67
68    public void setChoice(String choice) {
69      this.choice = choice;
70    }
71
72    public ResultSet getStudents() throws SQLException {
73      if (choice == null) {
74        if (titles.length == 0)
75          return null;
76        else
77          studentStatement.setString(1, titles[0]);
78      }
79      else {
80        studentStatement.setString(1, choice); // Set course title
81      }
82
83      // Get students for the specified course
84      return studentStatement.executeQuery();
85    }
86  }
```

`titles` array

get students

set a default course

set a course

return students

We use the same MySQL database **javabook** created in Chapter 34, "Java Database Programming." The scope for this managed bean is **application**. The bean is created when the project is launched from the server. The `initializeJdbc` method loads the JDBC driver for MySQL (lines 23 and 24), connects to the MySQL database (lines 27 and 28), creates statement for obtaining course titles (lines 31 and 32), and creates a statement for obtaining the student information for the specified course (lines 45–53). Lines 31–42 execute the statement for obtaining course titles and store them in array `titles`.

The `getStudents()` method returns a `ResultSet` that consists of all students enrolled in the specified course (lines 72–85). The choice for the title is set in the statement to obtain the student for the specified title (line 80). If choice is `null`, the first title in the titles array is set in the statement (line 77). If no titles in the course, `getStudents()` returns `null` (line 75).

add MySQL in the Libraries node

TIP

In order to use the MySQL database from this project, you have to add the MySQL JDBC driver from the Libraries node in the Project pane in NetBeans.

LISTING 39.14 `DisplayStudent.xhtml`

style sheet

bind choice

titles

display button

bind result set

rowClasses

headerClass

styleClass

ssn column

firstName column

mi column

lastName column

phone column

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
  <h:head>
    <title>Display Student</title>
    <h:outputStylesheet name="tablestyle.css"/>
  </h:head>
  <h:body>
    <h:form>
      <h:outputLabel value="Choose a Course: " />
      <h:selectOneMenu value="#{courseName.choice}">
        <f:selectItems value="#{courseName.titles}" />
      </h:selectOneMenu>

      <h:commandButton style="margin-left: 20px"
                       value="Display Students" />

      <br /> <br />
      <h:dataTable value="#{courseName.students}" var="student"
                   rowClasses="oddTableRow, evenTableRow"
                   headerClass="tableHeader"
                   styleClass="table">
        <h:column>
          <f:facet name="header">SSN</f:facet>
          #{student.ssn}
        </h:column>

        <h:column>
          <f:facet name="header">First Name</f:facet>
          #{student.firstName}
        </h:column>

        <h:column>
          <f:facet name="header">MI</f:facet>
          #{student.mi}
        </h:column>

        <h:column>
          <f:facet name="header">Last Name</f:facet>
          #{student.lastName}
        </h:column>

        <h:column>
          <f:facet name="header">Phone</f:facet>
          #{student.phone}
        </h:column>

        <h:column>
```

```
52            <f:facet name="header">Birth Date</f:facet>
53            #{student.birthDate}
54          </h:column>
55
56          <h:column>
57            <f:facet name="header">Dept</f:facet>
58            #{student.deptId}
59          </h:column>
60        </h:dataTable>
61      </h:form>
62    </h:body>
63  </html>
```

birthDate column

deptId column

Line 9 specifies that the style sheet tablestyle.css created in Step 3 is used in this XMTHL file. The `rowClasses = "oddTableRow, evenTableRow"` attribute specifies the style applied to the rows alternately using `oddTableRow` and `evenTableRow` (line 23). The `headerClasses = "tableHeader"` attribute specifies that the `tableHeader` class is used for header style (line 24). The `styleClasses = "table"` attribute specifies that the `table` class is used for the style of all other elements in the table (line 25).

Line 14 binds the `choice` property in the `courseName` bean with the combo box. The selection values in the combo box are bound with the `titles` array property (line 15).

Line 22 binds the table value with a database result set using the attribute `value="#{courseName.students}"`. The `var="student"` attribute associates a row in the result set with `student`. Lines 26–59 specify the column values using `student.ssn` (line 28), `student.firstName` (line 33), `student.mi` (line 38), `student.lastName` (line 33), `student.phone` (line 48), `student.birthDate` (line 53), and `student.deptId` (line 58).

LISTING 39.15 tablestyle.css

```
/* Style for table */
.tableHeader {
  font-family:"Trebuchet MS", Arial, Helvetica, sans-serif;
  border-collapse:collapse;
  font-size:1.1em;
  text-align:left;
  padding-top:5px;
  padding-bottom:4px;
  background-color:#A7C942;
  color:white;
  border:1px solid #98bf21;
}

.oddTableRow {
  border:1px solid #98bf21;
}

.evenTableRow {
  background-color: #eeeeee;
  font-size:1em;

  padding:3px 7px 2px 7px;

  color:#000000;
  background-color:#EAF2D3;
}

.table {
  border:1px solid green;
}
```

tableHeader

oddTableRow

evenTableRow

table

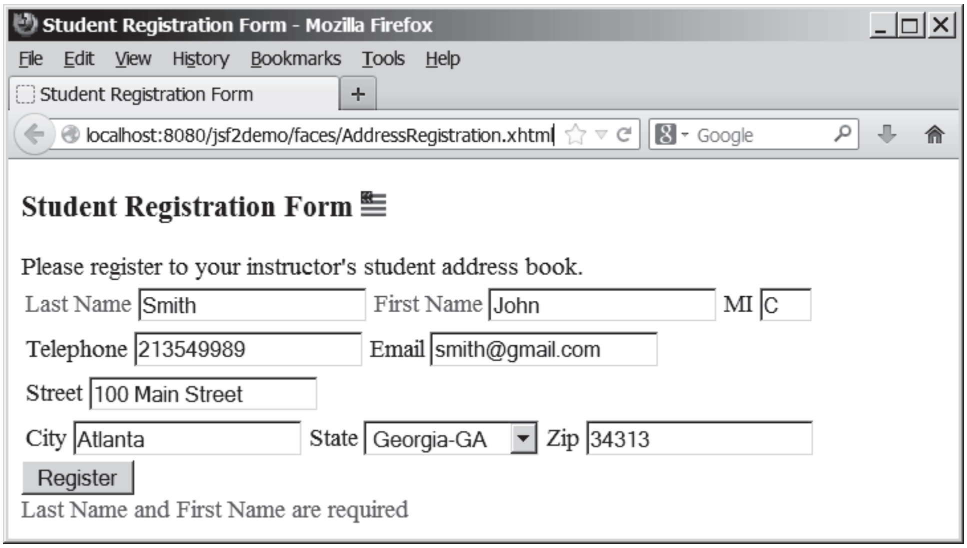
The style sheet file defines the style classes **`tableHeader`** (line 2) for table header style, **`oddTableRow`** for odd table rows (line 14), **`evenTableRow`** for even table rows (line 18), and table for all other table elements (line 28).

39.9 Opening New JSF Pages

Key Point

You can open new JSF pages from the current JSF pages.

All the examples you have seen so far use only one JSF page in a project. Suppose you want to register student information to the database. The application first displays the page as shown in Figures 39.22 to collect student information. After the user enters the information and clicks the *Submit* button, a new page is displayed to ask the user to confirm the input, as shown in Figure 39.23. If the user clicks the *Confirm* button, the data are stored into the database and the status page is displayed, as shown in Figure 39.24. If the user clicks the *Go Back* button, it goes back to the first page.


Student Registration Form - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Student Registration Form
+
localhost:8080/jsf2demo/faces/AddressRegistration.xhtml
Google
Register

Student Registration Form

Please register to your instructor's student address book.

Last Name Smith First Name John MI C

Telephone 213549989 Email smith@gmail.com

Street 100 Main Street

City Atlanta State Georgia-GA Zip 34313

Last Name and First Name are required

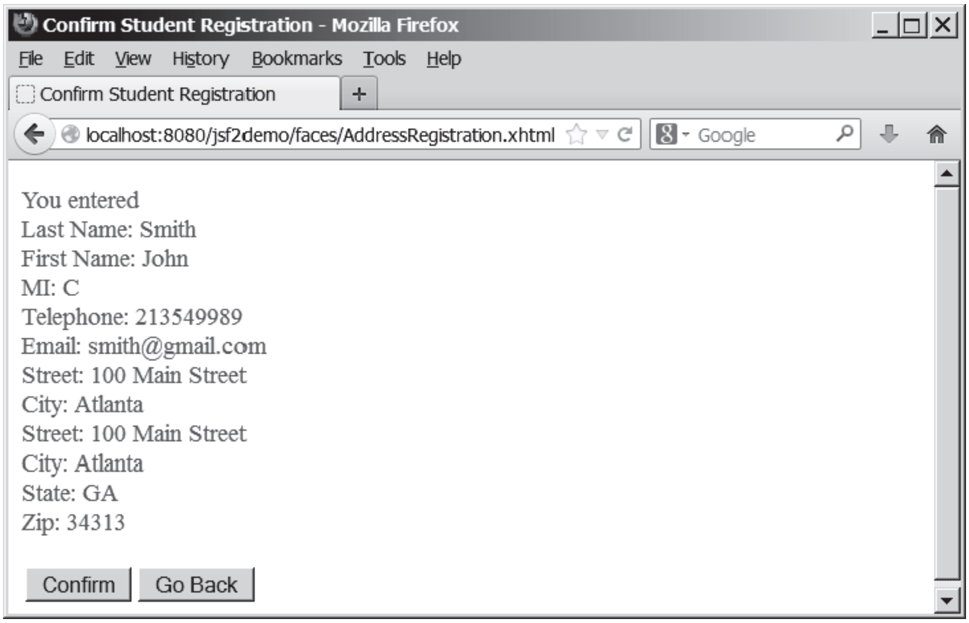
FIGURE 39.22 This page lets the user enter input.

For this project, you need to create three JSF pages named AddressRegistration.xhtml, ConfirmAddress.xhtml, and AddressStoredStatus.xhtml in Listings 39.16–39.18. The project starts with AddressRegistration.xhtml. When clicking the *Submit* button, the action for the button returns "ConfirmAddress" if the last name and first name are not empty, which causes ConfirmAddress.xhtml to be displayed. When clicking the *Confirm* button, the status page AddressStoredStatus is displayed. When clicking the *Go Back* button, the first page AddressRegistration is now displayed.

LISTING 39.16 `AddressRegistration.xhtml`

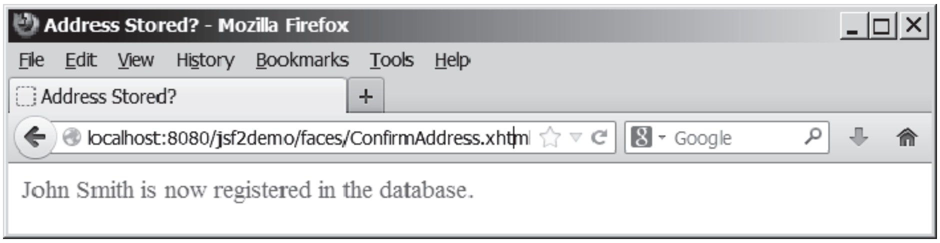
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
  <h:head>
    <title>Student Registration Form</title>
```

jsf core namespace


Confirm Student Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Confirm Student Registration
localhost:8080/jsf2demo/faces/AddressRegistration.xhtml
Google
Confirm
Go Back

You entered
Last Name: Smith
First Name: John
MI: C
Telephone: 213549989
Email: smith@gmail.com
Street: 100 Main Street
City: Atlanta
Street: 100 Main Street
City: Atlanta
State: GA
Zip: 34313

FIGURE 39.23 This page lets the user confirm the input.


Address Stored? - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Address Stored?
localhost:8080/jsf2demo/faces/ConfirmAddress.xhtml
Google

John Smith is now registered in the database.

FIGURE 39.24 This page displays the status of the user input.

```
  </h:head>
  <h:body>
    <h:form>
      <!-- Use h:graphicImage -->
      <h3>Student Registration Form
        <h:graphicImage name="usIcon.gif" library="image"/>
      </h3>

      Please register to your instructor's student address book.
      <!-- Use h:panelGrid -->
      <h:panelGrid columns="6">
        <h:outputLabel value="Last Name" style="color:red"/>
        <h:inputText id="lastNameInputText"
                     value="#{addressRegistration.lastName}"/>
        <h:outputLabel value="First Name" style="color:red"/>
        <h:inputText id="firstNameInputText"
                     value="#{addressRegistration.firstName}"/>
        <h:outputLabel value="MI" />
        <h:inputText id="miInputText" size="1"
                     value="#{addressRegistration.mi}"/>
      </h:panelGrid>

```

bind `lastName`

bind `firstName`

bind `mi`

```
      <h:panelGrid columns="4">
        <h:outputLabel value="Telephone"/>
        <h:inputText id="telephoneInputText"
                     value="#{addressRegistration.telephone}"/>
        <h:outputLabel value="Email"/>
        <h:inputText id="emailInputText"
                     value="#{addressRegistration.email}"/>
      </h:panelGrid>

      <h:panelGrid columns="4">
        <h:outputLabel value="Street"/>
        <h:inputText id="streetInputText"
                     value="#{addressRegistration.street}"/>
      </h:panelGrid>

      <h:panelGrid columns="6">
        <h:outputLabel value="City"/>
        <h:inputText id="cityInputText"
                     value="#{addressRegistration.city}"/>
        <h:outputLabel value="State"/>
        <h:selectOneMenu id="stateSelectOneMenu"
                         value="#{addressRegistration.state}">
          <f:selectItem itemLabel="Georgia-GA" itemValue="GA" />
          <f:selectItem itemLabel="Oklahoma-OK" itemValue="OK" />
          <f:selectItem itemLabel="Indiana-IN" itemValue="IN"/>
        </h:selectOneMenu>
        <h:outputLabel value="Zip"/>
        <h:inputText id="zipInputText"
                     value="#{addressRegistration.zip}"/>
      </h:panelGrid>

      <!-- Use command button -->
      <h:commandButton value="Register"
                       action="#{addressRegistration.processSubmit()}"/>
      <br />
      <h:outputText escape="false" style="color:red"
                    value="#{addressRegistration.requiredFields}" />
    </h:form>
  </h:body>
</html>
```

bind telephone

bind email

bind street

bind city

bind state

bind zip

process register

Listing 39.17 `ConfirmAddress.xhtml`

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>Confirm Student Registration</title>
  </h:head>
  <h:body>
    <h:form>
      <h:outputText escape="false" style="color:red"
                    value="#{registration1.input}" />
      <h:panelGrid columns="2">
      <h:commandButton value="Confirm"
         action = "#{registration1.storeStudent()}"/>
      <h:commandButton value="Go Back"
         action = "AddressRegistration"/>
```

process confirm

go to AddressRegistration page

```
      </h:panelGrid>
    </h:form>
  </h:body>
</html>
```

LISTING 39.18 `AddressStoredStatus.xhtml`

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>Address Stored?</title>
  </h:head>
  <h:body>
    <h:form>
      <h:outputText escape="false" style="color:green"
                    value="#{registration1.status}" />
    </h:form>
  </h:body>
</html>
```

display status

LISTING 39.19 `AddressRegistrationJSFBean.java`

```
package jsf2demo;

import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.sql.*;
import java.io.Serializable;

@Named(value = "addressRegistration")
@SessionScoped
public class AddressRegistrationJSFBean implements Serializable {
  private String lastName;
  private String firstName;
  private String mi;
  private String telephone;
  private String email;
  private String street;
  private String city;
  private String state;
  private String zip;
  private String status = "Nothing stored";
  // Use a prepared statement to store a student into the database
  private PreparedStatement pstmt;

  public AddressRegistrationJSFBean() {
    initializeJdbc();
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

```

managed bean

session scope

property `lastName`

initialize database

```
  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getMi() {
    return mi;
  }

  public void setMi(String mi) {
    this.mi = mi;
  }

  public String getTelephone() {
    return telephone;
  }

  public void setTelephone(String telephone) {
    this.telephone = telephone;
  }

  public String getEmail() {
    return email;
  }

  public void setEmail(String email) {
    this.email = email;
  }

  public String getStreet() {
    return street;
  }

  public void setStreet(String street) {
    this.street = street;
  }

  public String getCity() {
    return city;
  }

  public void setCity(String city) {
    this.city = city;
  }

  public String getState() {
    return state;
  }

  public void setState(String state) {
    this.state = state;
  }

  public String getZip() {
    return zip;
  }

  public void setZip(String zip) {
    this.zip = zip;
  }

  private boolean isRquiredFieldsFilled() {
    return !(lastName == null || firstName == null
           || lastName.trim().length() == 0
           || firstName.trim().length() == 0);
  }

  public String processSubmit() {
    if (isRquiredFieldsFilled())
      return "ConfirmAddress";
    else
      return "";
  }

  public String getRequiredFields() {
    if (isRquiredFieldsFilled())
      return "";
    else
      return "Last Name and First Name are required";
  }

  public String getInput() {
    return "<p style=\"color:red\">You entered <br />"
            + "Last Name: " + lastName + "<br />"
            + "First Name: " + firstName + "<br />"
            + "MI: " + mi + "<br />"
            + "Telephone: " + telephone + "<br />"
            + "Email: " + email + "<br />"
            + "Street: " + street + "<br />"
            + "City: " + city + "<br />"
            + "Street: " + street + "<br />"
            + "City: " + city + "<br />"
            + "State: " + state + "<br />"
            + "Zip: " + zip + "</p>";
  }

  /** Initialize database connection */
  private void initializeJdbc() {
    try {
      // Explicitly load a MySQL driver
      Class.forName("com.mysql.jdbc.Driver");
      System.out.println("Driver loaded");

      // Establish a connection
      Connection conn = DriverManager.getConnection(
             "jdbc:mysql://localhost/javabook", "scott", "tiger");

      // Create a Statement
      pstmt = conn.prepareStatement("insert into Address (lastName,"
             + " firstName, mi, telephone, email, street, city, "
             + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }

  /** Store an address to the database */
  public String storeStudent() {
```

go to a new page

check required fields

get input

store address

update status

go to a new page

```
158     try {
159       pstmt.setString(1, lastName);
160       pstmt.setString(2, firstName);
161       pstmt.setString(3, mi);
162       pstmt.setString(4, telephone);
163       pstmt.setString(5, email);
164       pstmt.setString(6, street);
165       pstmt.setString(7, city);
166       pstmt.setString(8, state);
167       pstmt.setString(9, zip);
168       pstmt.executeUpdate();
169       status = firstName + " " + lastName
170               + " is now registered in the database.";
171     }
172     catch (Exception ex) {
173       status = ex.getMessage();
174     }
175 
176     return "AddressStoredStatus";
177   }
178 
179   public String getStatus() {
180     return status;
181   }
182 }
```

A session-scoped managed bean must implement the `java.io.Serializable` interface. So, the `AddressRegistration` class is defined as a subtype of `java.io.Serializable`.

The action for the *Register* button in the `AddressRegistration` JSF page is `processSubmit()` (line 64 in AddressRegistration.xhtml). This method checks if last name and first name are not empty (lines 106–111 in AddressRegistrationJSFBean.java). If so, it returns a string `"ConfirmAddress"`, which causes the `ConfirmAddress` JSF page to be displayed.

The `ConfirmAddress` JSF page displays the data entered from the user (line 12 in ConfirmAddress.xhtml). The `getInput()` method (lines 120–133 in AddressRegistrationJSFBean.java) collects the input.

The action for the *Confirm* button in the `ConfirmAddress` JSF page is `storeStudent()` (line 15 in ConfirmAddress.xhtml). This method stores the address in the database (lines 157–177 in AddressRegistrationJSFBean.java) and returns a string `"AddressStoredStatus"`, which causes the `AddressStoredStatus` page to be displayed. The status message is displayed in this page (line 12 in AddressStoredStatus.xhtml).

The action for the *Go Back* button in the `ConfirmAddress` page is `"AddressRegistration"` (line 17 in ConfirmAddress.xhtml). This causes the `AddressRegistration` page to be displayed for the user to reenter the input.

The scope of the managed bean is session (line 9 AddressRegistrationJSFBean.java) so the multiple pages can share the same bean.

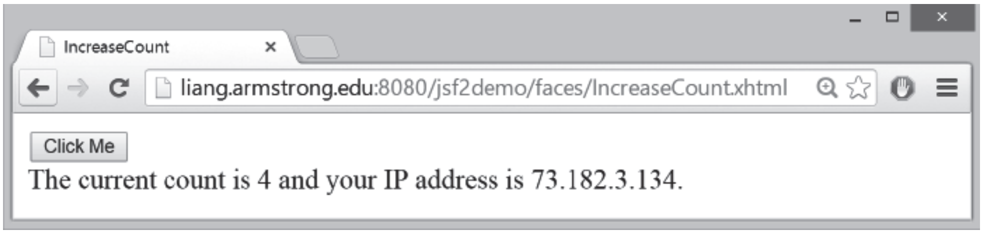
Note that this program loads the database driver explicitly (line 139 AddressRegistrationJSFBean.java). Sometimes, an IDE such as NetBeans is not able to find a suitable driver. Loading a driver explicitly can avoid this problem.

39.10 Contexts and Dependency Injection

Key Point

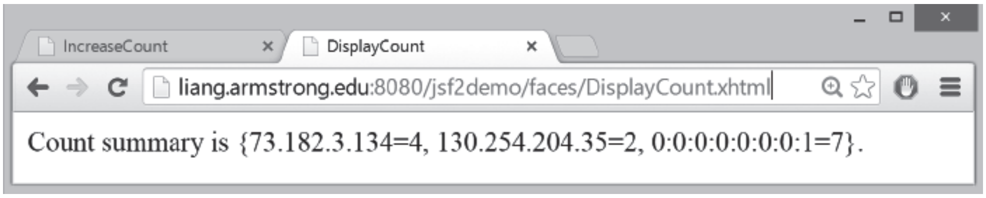
Contexts and dependency injection enables beans to be shared in multiple applications.

Contexts and dependency injection, short for *CDI*, allows multiple programs to share a bean. To illustrate the need for this, consider two simple Web pages and a server object named `track`. One page contains a button and a message that displays the number of times the button is clicked from the current IP address, as shown in Figure 39.25. When the button is clicked for the first time, the user's IP address along with count value 1 is stored in a map with the IP address as the key. When the button is clicked again, the count value for the IP address is increased in the map. The other page simply displays the total count from each IP address, as shown in Figure 39.26. The `Track` class is defined as shown in Listing 39.20.



The current count is 4 and your IP address is 73.182.3.134.

Figure 39.25 The count is updated when the Click Me button is clicked.



Count summary is {73.182.3.134=4, 130.254.204.35=2, 0:0:0:0:0:0:0:1=7}.

Figure 39.26 The count for each client IP Address is displayed.

Listing 39.20 Track.java

```
 1  package jsf2demo;
 2
 3  import java.util.HashMap;
 4  import java.util.Map;
 5  import javax.enterprise.context.ApplicationScoped;
 6
 7  @ApplicationScoped
 8  public class Track {
 9    private Map<String, Integer> map = new HashMap<>();
10
11    public void add(String ipAddress) {
12      map.put(ipAddress, map.containsKey(ipAddress) ?
13        map.get(ipAddress) + 1 : 1);
14    }
15
16    public int getCount(String ipAddress) {
17      return map.containsKey(ipAddress) ? map.get(ipAddress) : 0;
18    }
19
20    public String getAllCount() {
21      return "Count summary is " + map;
22    }
23  }
```

application scope
store counts
add or update count
return count
return all counts

A `Track` object uses a map to store an IP address and its count with IP address as a key (line 9). The `add` method (lines 11–14) adds an IP Address to the map. If the IP address is not in the map, a new entry is created for the IP Address with value `1`. Otherwise, the value for the IP address is incremented by `1` in the map. The `getCount` method (lines 16–18) returns the count for an IP address. If the IP address is not in the map, the method returns `0`. The `getAllCount` method (lines 20–22) simply returns a string that describes the counts for all IP address in the map.

We now create a page named **IncreaseCount.xhtml** (Listing 39.21) with a button for displaying the number of times a button is clicked on the client, and create a page named **DisplayCount.xhtml** (Listing 39.22) for displaying the counts from all clients.

Listing 39.21 IncreaseCount.xhtml

process a click
obtain count
obtain IP address

```
 1  <?xml version='1.0' encoding='UTF-8' ?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 4  <html xmlns="http://www.w3.org/1999/xhtml"
 5        xmlns:h="http://xmlns.jcp.org/jsf/html">
 6    <h:head>
 7      <title>IncreaseCount</title>
 8    </h:head>
 9    <h:body>
10      <h:form>
11        <h:commandButton
12          action="#{increaseCount.click()}" value="Click Me"/>
13        <br>The current count is #{increaseCount.getCount()} and your
14          IP address is #{increaseCount.getIpAddress()}.</br>
15      </h:form>
16    </h:body>
17  </html>
```

Listing 39.22 DisplayCount.xhtml

obtain all counts

```
 1  <?xml version='1.0' encoding='UTF-8' ?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 4  <html xmlns="http://www.w3.org/1999/xhtml"
 5        xmlns:h="http://xmlns.jcp.org/jsf/html">
 6    <h:head>
 7      <title>DisplayCount</title>
 8    </h:head>
 9    <h:body>
10      #{displayCount.getAllCount()}.
11    </h:body>
12  </html>
```

The `IncreaseCount` page uses the `increasCount` bean to process the click action (line 12), obtain the click count (line 13), and the client's IP address (line 14). The `DisplayCount` page uses the `displayCount` bean to obtain the count from all clients (line 10). Both `increasCount` and `displayCount` need to access the same `track` object. How can you create a `Track` object to be used by different objects? JSF supports context dependency injection (CDI) for injecting an object into a class using the `@Inject` annotation. Listing 39.23 gives the implementation for IncreaseCount.java and Listing 39.24 for DisplayCount.java.

Listing 39.23 IncreaseCount.java

```
package jsf2demo;

import javax.enterprise.context.SessionScoped;
import javax.inject.Named;
import javax.faces.context.FacesContext;
import javax.inject.Inject;
import javax.servlet.http.HttpServletRequest;

@Named(value = "increaseCount")
```

```
10 @SessionScoped
11 public class IncreaseCount implements java.io.Serializable {
12   @Inject private Track track;
13   private String ipAddress;
14 
15   public IncreaseCount() {
16     HttpServletRequest request = (HttpServletRequest)FacesContext
17       .getCurrentInstance().getExternalContext().getRequest();
18     this.ipAddress = request.getRemoteAddr();
19   }
20 
21   public void click() {
22     track.add(ipAddress);
23   }
24 
25   public String getIpAddress() {
26     return ipAddress;
27   }
28 
29   public int getCount() {
30     return track.getCount(ipAddress);
31   }
32 }
```

session scope

inject track

increase count

obtain client's IP

add an IP address

count for an IP

LISTING 39.24 DisplayCount.java

```
 1 package jsf2demo;
 2 
 3 import javax.enterprise.context.ApplicationScoped;
 4 import javax.inject.Named;
 5 import javax.inject.Inject;
 6 
 7 @Named(value = "displayCount")
 8 @ApplicationScoped
 9 public class DisplayCount {
10   @Inject private Track track;
11 
12   public String getAllCount() {
13     return track.getAllCount();
14   }
15 }
```

application scope

inject track

obtain all counts

The `@Inject` annotation in line 12 of **IncreaseCount.java** and line 10 of **DisplayCount.java** injects a `Track` object. This `Track` object is created by the Java server container. The `track` data fields in both classes refer to this object.

In **IncreaseCount.java**, the constructor obtains the IP address of a client (lines 16 and 17) and sets it in the data field `ipAddress` (line 18). The `click` method adds the `ipAddress` to the map in the track object (line 22).

Note that the scope for `Track` and `DisplayCount` is `ApplicationScoped` since these two objects are created once for the entire application. However, the scope for `IncreaseCount` is `SessionScoped` since each session has its own IP Address.

KEY TERMS

application scope 39-21
contexts and dependency injection (CDI) 39-40
JavaBean 39-5
request scope 39-21
scope 39-21
session scope 39-21
view scope 39-21

CHAPTER SUMMARY

1. JSF enables you to completely separate Java code from HTML.

2. A `facelet` is an XHTML page that mixes JSF tags with XHTML tags.

3. JSF applications are developed using the Model-View-Controller (MVC) architecture, which separates the application's data (contained in the model) from the graphical presentation (the view).

4. The controller is the JSF framework that is responsible for coordinating interactions between view and the model.

5. In JSF, the facelets are the view for presenting data. Data are obtained from Java objects. Objects are defined using Java classes.

6. In JSF, the objects that are accessed from a facelet are JavaBeans objects.

7. The JSF expression can either use the property name or invoke the method to obtain the current time.

8. JSF provides many elements for displaying GUI components. The tags with the `h` prefix are in the JSF HTML Tag library. The tags with the `f` prefix are in the JSF Core Tag library.

9. You can specify the JavaBeans objects at the application scope, session scope, view scope, or request scope.

10. The view scope keeps the bean alive as long as you stay on the view. The view scope is between session and request scopes.

11. JSF provides several convenient and powerful ways for input validation. You can use the standard validator tags in the JSF Core Tag Library or create custom validators.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

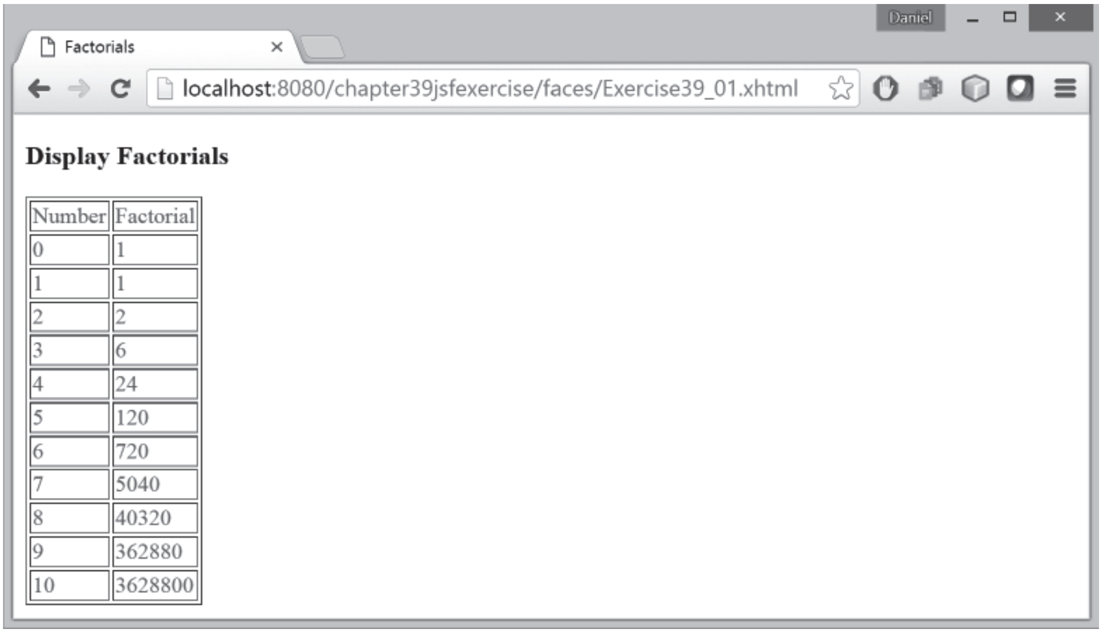
PROGRAMMING EXERCISES

***39.1** *(Factorial table in JSF)* Write a JSF page that displays a factorial page as shown in Figure 39.27. Display the table in an `h:outputText` component. Set its `escape` property to `false` to display it as HTML contents.

***39.2** (*Multiplication table*) Write a JSF page that displays a multiplication table as shown in Figure 39.28.

***39.3** (*Calculate tax*) Write a JSF page to let the user enter taxable income and filing status, as shown in Figure 39.29a. Clicking the *Compute Tax* button computes and displays the tax, as shown in Figure 39.29b. Use the `computeTax` method introduced in Listing 3.5, ComputeTax.java, to compute tax.

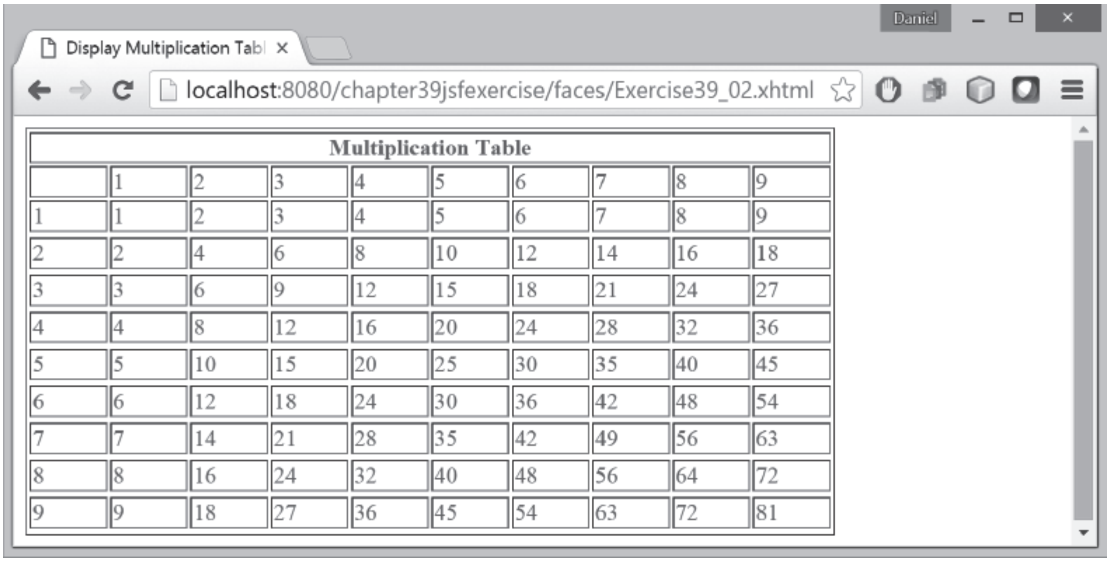
***39.4** (*Calculate loan*) Write a JSF page that lets the user enter loan amount, interest rate, and number of years, as shown in Figure 39.30a. Click the *Compute Loan Payment* button to compute and display the monthly and total loan payments, as shown in Figure 39.30b. Use the `Loan` class given in Listing 10.2, Loan.java, to compute the monthly and total payments.


Daniel
Factorials
localhost:8080/chapter39jsfexercise/faces/Exercise39_01.xhtml

Display Factorials

Number	Factorial
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

FIGURE 39.27 The JSF page displays factorials for the numbers from 0 to 10 in a table.


Daniel
Display Multiplication Tabl
localhost:8080/chapter39jsfexercise/faces/Exercise39_02.xhtml

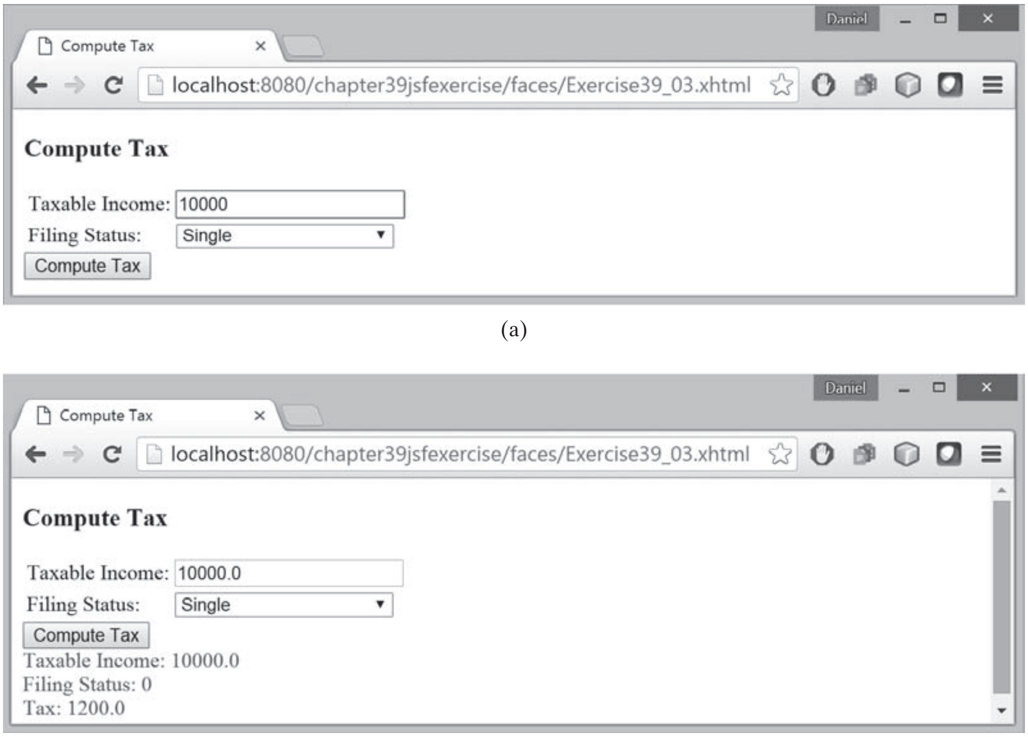
Multiplication Table									
	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

FIGURE 39.28 The JSF page displays the multiplication table.

***39.5** (*Addition quiz*) Write a JSF program that generates addition quizzes randomly, as shown in Figure 39.31a. After the user answers all questions, it displays the result, as shown in Figure 39.31b.

***39.6** (*Large factorial*) Rewrite Exercise 39.1 to handle large factorial as shown in Figure 39.32. Use the `BigInteger` class introduced in Section 10.9.

***39.7** (*Guess birthday*) Listing 4.3, GuessBirthday.java, gives a program for guessing a birthday. Write a JSF program that displays five sets of numbers, as shown in Figure 39.33a. After the user checks the appropriate boxes and clicks the *Guess Birthday* button, the program displays the birthday, as shown in Figure 39.33b.


Compute Tax
Daniel
localhost:8080/chapter39jsfexercise/faces/Exercise39_03.xhtml
Compute Tax

(a)

Compute Tax

Taxable Income: 10000

Filing Status: Single

(b)

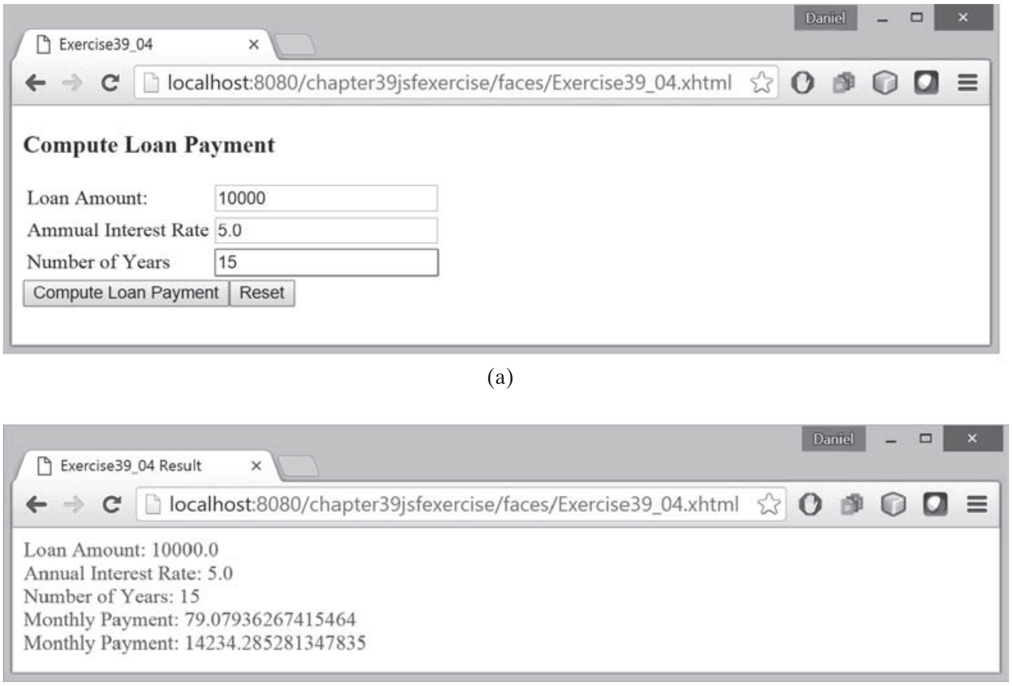
Compute Tax

Taxable Income: 10000.0

Filing Status: Single

```
Taxable Income: 10000.0
Filing Status: 0
Tax: 1200.0
```

FIGURE 39.29 The JSF page computes the tax.


Exercise39_04
Exercise39_04 Result
Daniel
localhost:8080/chapter39jsfexercise/faces/Exercise39_04.xhtml
Compute Loan Payment
Reset

(a)

Compute Loan Payment

Loan Amount: 10000

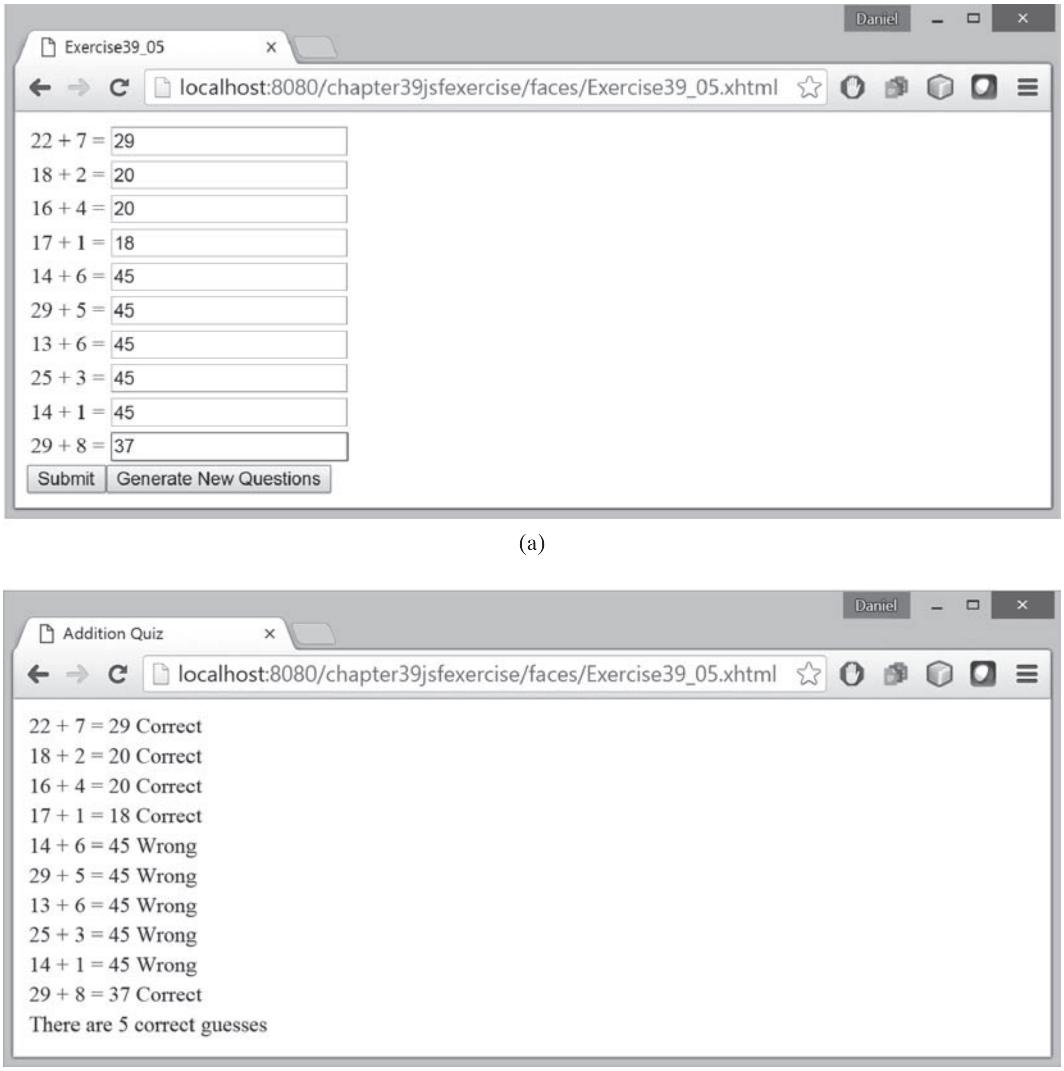
Ammual Interest Rate 5.0

Number of Years 15

(b)

```
Loan Amount: 10000.0
Annual Interest Rate: 5.0
Number of Years: 15
Monthly Payment: 79.07936267415464
Monthly Payment: 14234.285281347835
```

FIGURE 39.30 The JSF page computes the loan payment.


Exercise39_05
localhost:8080/chapter39jsfexercise/faces/Exercise39_05.xhtml
Daniel

22 + 7 = 29
18 + 2 = 20
16 + 4 = 20
17 + 1 = 18
14 + 6 = 45
29 + 5 = 45
13 + 6 = 45
25 + 3 = 45
14 + 1 = 45
29 + 8 = 37

Submit
Generate New Questions

(a)

Addition Quiz
localhost:8080/chapter39jsfexercise/faces/Exercise39_05.xhtml
Daniel

22 + 7 = 29 Correct
18 + 2 = 20 Correct
16 + 4 = 20 Correct
17 + 1 = 18 Correct
14 + 6 = 45 Wrong
29 + 5 = 45 Wrong
13 + 6 = 45 Wrong
25 + 3 = 45 Wrong
14 + 1 = 45 Wrong
29 + 8 = 37 Correct
There are 5 correct guesses

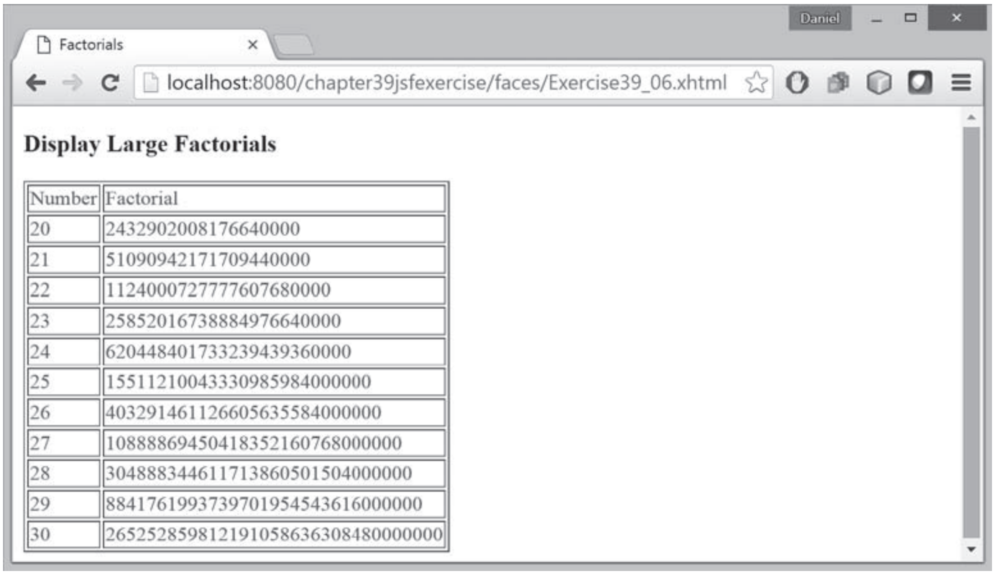
(b)

FIGURE 39.31 The program displays addition questions in (a) and answers in (b).

***39.8** (*Guess capitals*) Write a JSF that prompts the user to enter a capital for a state, as shown in Figure 39.34a. Upon receiving the user input, the program reports whether the answer is correct, as shown in Figure 39.34b. You can click the *Next* button to display another question. You can use a two-dimensional array to store the states and capitals, as proposed in Exercise 8.37. Create a list from the array and apply the `shuffle` method to reorder the list so the questions will appear in random order.

***39.9** (*Access and update a `Staff` table*) Write a JSF program that views, inserts, and updates staff information stored in a database, as shown in Figure 39.35. The view button displays a record with a specified ID. The `Staff` table is created as follows:

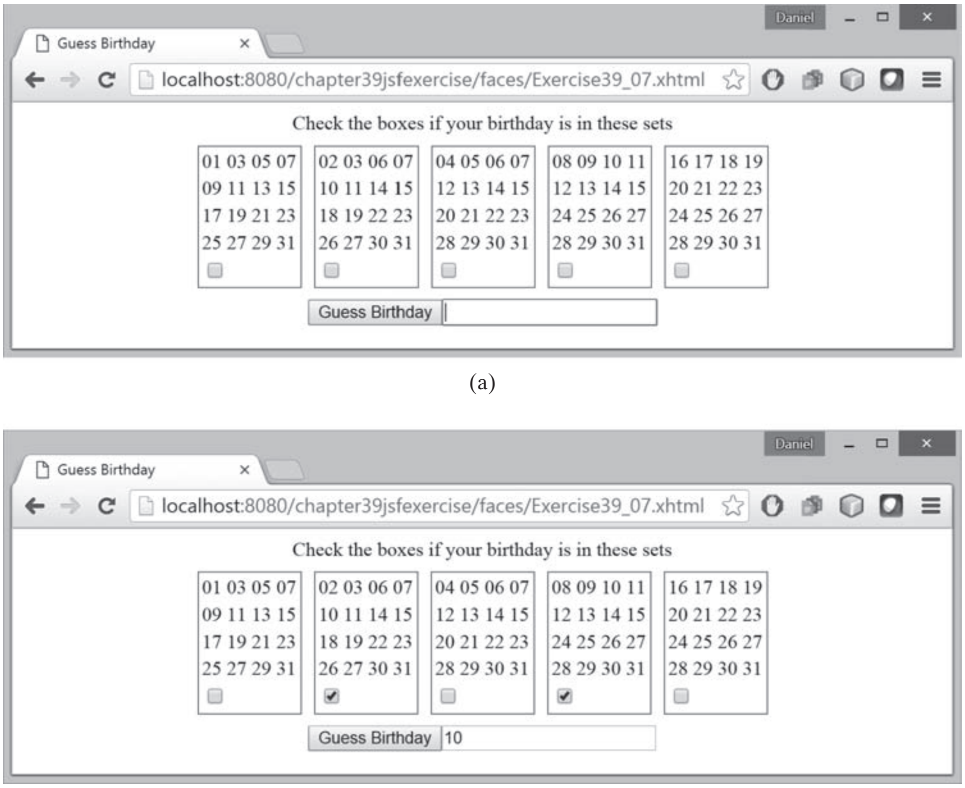
```
create table Staff (
  id char(9) not null,
  lastName varchar(15),
  firstName varchar(15),
```


Factorials
localhost:8080/chapter39jsfexercise/faces/Exercise39_06.xhtml
Daniel

Display Large Factorials

Number	Factorial
20	2432902008176640000
21	51090942171709440000
22	1124000727777607680000
23	25852016738884976640000
24	620448401733239439360000
25	15511210043330985984000000
26	403291461126605635584000000
27	10888869450418352160768000000
28	304888344611713860501504000000
29	8841761993739701954543616000000
30	265252859812191058636308480000000

Figure 39.32 The JSF page displays factorials for the numbers from 10 to 20 in a table.


Guess Birthday
localhost:8080/chapter39jsfexercise/faces/Exercise39_07.xhtml
Daniel
Guess Birthday

Check the boxes if your birthday is in these sets

01 03 05 07 09 11 13 15 17 19 21 23 25 27 29 31 ☐	02 03 06 07 10 11 14 15 18 19 22 23 26 27 30 31 ☐	04 05 06 07 12 13 14 15 20 21 22 23 28 29 30 31 ☐	08 09 10 11 12 13 14 15 24 25 26 27 28 29 30 31 ☐	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 ☐

(a)

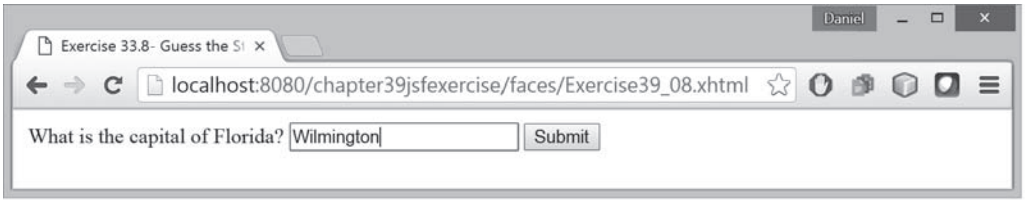
Check the boxes if your birthday is in these sets

01 03 05 07 09 11 13 15 17 19 21 23 25 27 29 31 ☐	02 03 06 07 10 11 14 15 18 19 22 23 26 27 30 31 ☑	04 05 06 07 12 13 14 15 20 21 22 23 28 29 30 31 ☐	08 09 10 11 12 13 14 15 24 25 26 27 28 29 30 31 ☑	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 ☐

Guess Birthday: 10

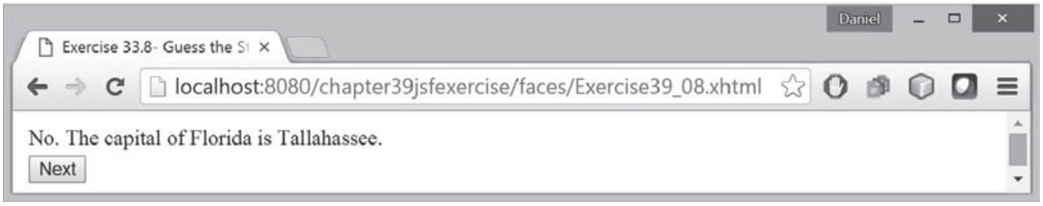
(b)

Figure 39.33 (a) The program displays five sets of numbers for the user to check the boxes. (b) The program displays the date.


Daniel
Exercise 33.8- Guess the St
localhost:8080/chapter39jsfexercise/faces/Exercise39_08.xhtml
Submit

What is the capital of Florida? Wilmington

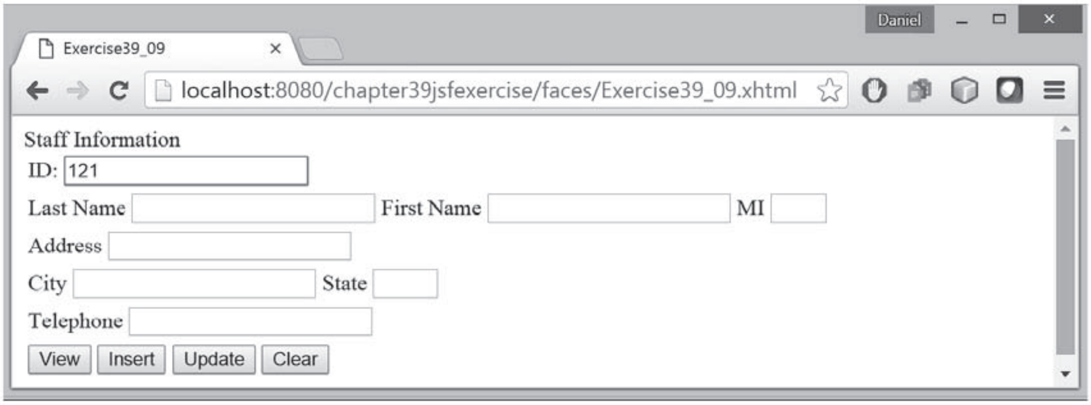
(a)


Daniel
Exercise 33.8- Guess the St
localhost:8080/chapter39jsfexercise/faces/Exercise39_08.xhtml
Next

No. The capital of Florida is Tallahassee.

(b)

Figure 39.34 (a) The program displays a question. (b) The program displays the answer to the question.


Daniel
Exercise39_09
localhost:8080/chapter39jsfexercise/faces/Exercise39_09.xhtml
View
Insert
Update
Clear

Staff Information

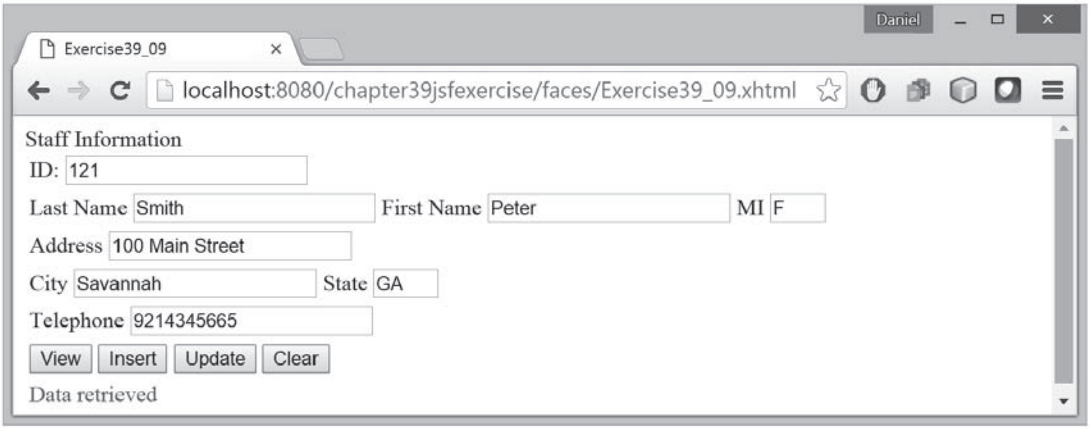
ID: 121

Last Name First Name MI

Address

City State

Telephone


Daniel
Exercise39_09
localhost:8080/chapter39jsfexercise/faces/Exercise39_09.xhtml
View
Insert
Update
Clear

Staff Information

ID: 121

Last Name Smith First Name Peter MI F

Address 100 Main Street

City Savannah State GA

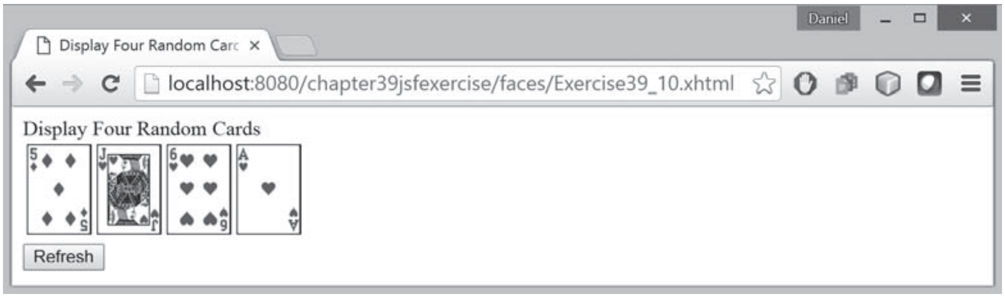
Telephone 9214345665

Data retrieved

Figure 39.35 The web page lets you view, insert, and update staff information.

```
    mi char(1),
    address varchar(20),
    city varchar(20),
    state char(2),
    telephone char(10),
    email varchar(40),
    primary key (id)
  );
```

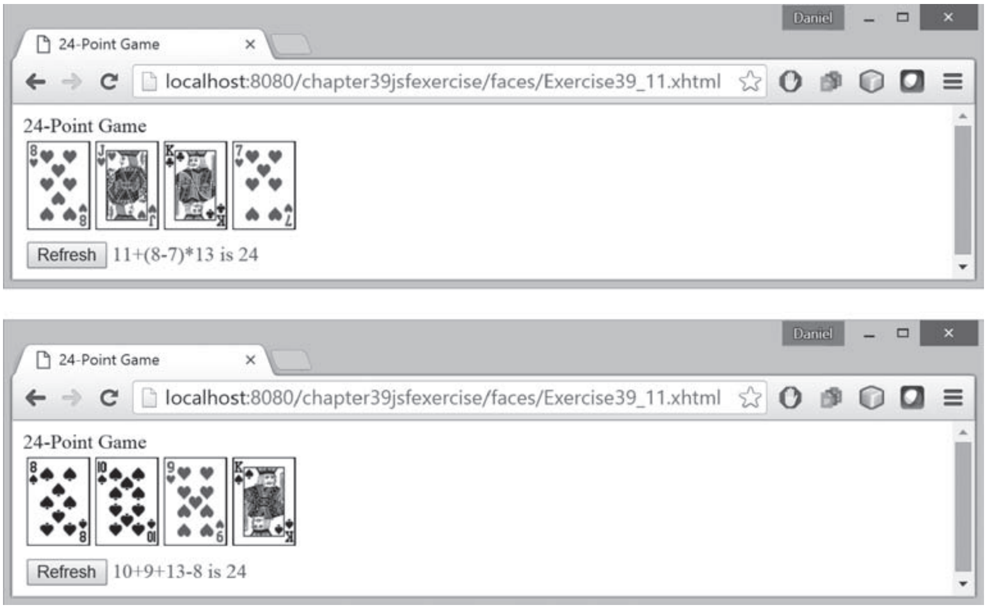
***39.10** (*Random cards*) Write a JSF that displays four random cards from a deck of 52 cards, as shown in Figure 39.36. When the user clicks the *Refresh* button, four new random cards are displayed.


Daniel
Display Four Random Carc
localhost:8080/chapter39jsfexercise/faces/Exercise39_10.xhtml
Refresh

Display Four Random Cards

FIGURE 39.36 This JSF application displays four random cards.

*****39.11** (*Game: the 24-point card game*) Rewrite Exercise 20.13 using JSF, as shown in Figure 39.37. Upon clicking the *Refresh* button, the program displays four random cards and displays an expression if a 24-point solution exists. Otherwise, it displays `No solution`.


Daniel
24-Point Game
localhost:8080/chapter39jsfexercise/faces/Exercise39_11.xhtml
Refresh
Daniel
24-Point Game
localhost:8080/chapter39jsfexercise/faces/Exercise39_11.xhtml
Refresh

24-Point Game

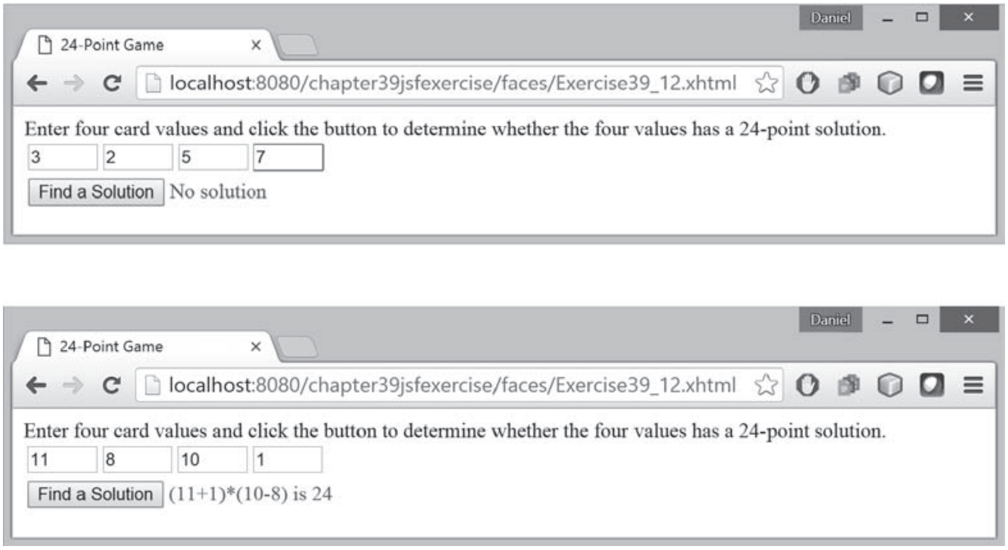
11+(8-7)*13 is 24

24-Point Game

10+9+13-8 is 24

FIGURE 39.37 The JSF application solves a 24-Point card game.

*****39.12** (*Game: the 24-point card game*) Rewrite Exercise 20.17 using JSF, as shown in Figure 39.38. The program lets the user enter four card values and finds a solution upon clicking the *Find a Solution* button.


Daniel
24-Point Game
localhost:8080/chapter39jsfexercise/faces/Exercise39_12.xhtml
Find a Solution

Enter four card values and click the button to determine whether the four values has a 24-point solution.

3 2 5 7

No solution

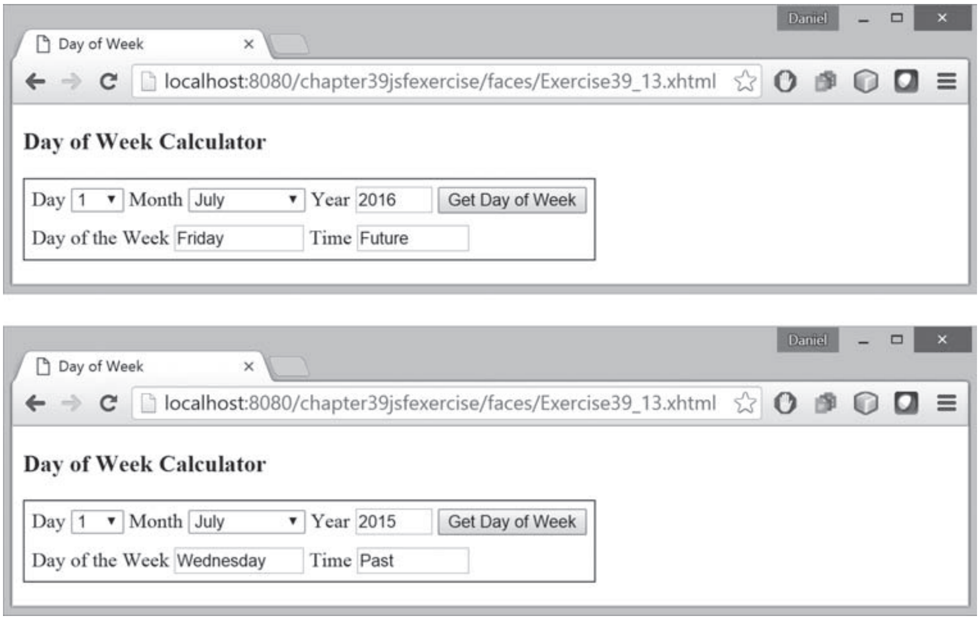
Enter four card values and click the button to determine whether the four values has a 24-point solution.

11 8 10 1

(11+1)*(10-8) is 24

FIGURE 39.38 The user enters four numbers and the program finds a solution.

***39.13** (*Day of week*) Write a program that displays the day of the week for a given day, month, and year, as shown in Figure 39.39. The program lets the user select a day, month, and year, and click the *Get Day of Week* button to display the day of week. The Time field displays Future if it is a future day or Past otherwise. Use the Zeller's congruence to find the day of the week (see Programming Exercise 3.21).


Daniel
Day of Week
localhost:8080/chapter39jsfexercise/faces/Exercise39_13.xhtml
Get Day of Week

Day of Week Calculator

Day 1 Month July Year 2016

Day of the Week Friday Time Future

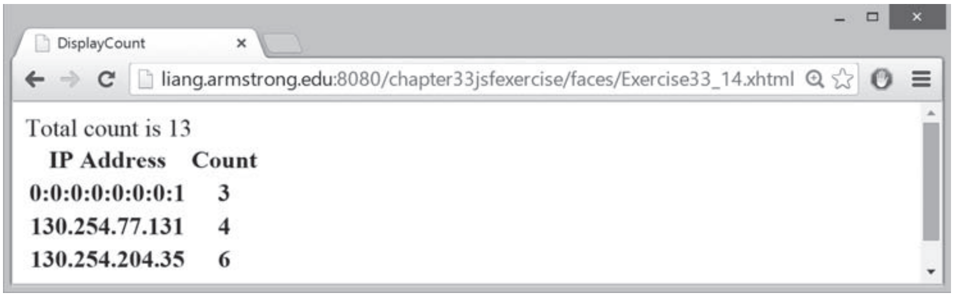
Day of Week Calculator

Day 1 Month July Year 2015

Day of the Week Wednesday Time Past

FIGURE 39.39 The user enters a day, month, and year and the program finds the day of the week.

*39.14 (*Display total count*) Revise Listing 39.22 DisplayCount.xhmt to display the total count of the button clicks form all clients and display the client's IP address and counts in increasing order of the counts, as shown in Figure 39.40.


DisplayCount
liang.armstrong.edu:8080/chapter33jsfexercise/faces/Exercise33_14.xhtml

Total count is 13

IP Address	Count
0:0:0:0:0:0:0:1	3
130.254.77.131	4
130.254.204.35	6

FIGURE 39.40 The total counts and individual client counts are displayed.

CHAPTER 40

REMOTE METHOD INVOCATION

Objectives

- To explain how RMI works (§40.2).
- To describe the process of developing RMI applications (§40.3).
- To distinguish between RMI and socket-level programming (§40.4).
- To develop three-tier applications using RMI (§40.5).
- To use callbacks to develop interactive applications (§40.6).

40.1 Introduction

Remote Method Invocation is a high-level Java API for Java network programming.

Remote Method Invocation (RMI) provides a framework for building distributed Java systems. Using RMI, a Java object on one system can invoke a method in an object on another system on the network. A *distributed Java system* can be defined as a collection of cooperative distributed objects on the network. In this chapter, you will learn how to use RMI to create useful distributed applications.

40.2 RMI Basics

RMI enables you to access a remote object and invoke its methods.

RMI is the Java Distributed Object Model for facilitating communications among distributed objects. RMI is a high-level API built on top of sockets. Socket-level programming allows you to pass data through sockets among computers. RMI enables you also to invoke methods in a remote object. Remote objects can be manipulated as if they were residing on the local host. The transmission of data among different machines is handled by the JVM transparently.

In many ways, RMI is an evolution of the client/server architecture. A *client* is a component that issues requests for services, and a *server* is a component that delivers the requested services. Like the client/server architecture, RMI maintains the notion of clients and servers, but the RMI approach is more flexible.

- An RMI component can act as both a client and a server, depending on the scenario in question.
- An RMI system can pass functionality from a server to a client, and vice versa. Typically a client/server system only passes data back and forth between server and client.

40.2.1 How Does RMI Work?

All the objects you have used before this chapter are called *local objects*. *Local objects* are accessible only within the local host. Objects that are accessible from a remote host are called *remote objects*. For an object to be invoked remotely, it must be defined in a Java interface accessible to both the server and the client. Furthermore, the interface must extend the `java.rmi.Remote` interface. Like the `java.io.Serializable` interface, `java.rmi.Remote` is a marker interface that contains no constants or methods. It is used only to identify remote objects.

The key components of the RMI architecture are listed below (see Figure 40.1):

- **Server object interface:** A subinterface of `java.rmi.Remote` that defines the methods for the server object.
- **Server class:** A class that implements the remote object interface.
- **Server object:** An instance of the server class.
- **RMI registry:** A utility that registers remote objects and provides naming services for locating objects.
- **Client program:** A program that invokes the methods in the remote server object.
- **Server stub:** An object that resides on the client host and serves as a surrogate for the remote server object.
- **Server skeleton:** An object that resides on the server host and communicates with the stub and the actual server object.

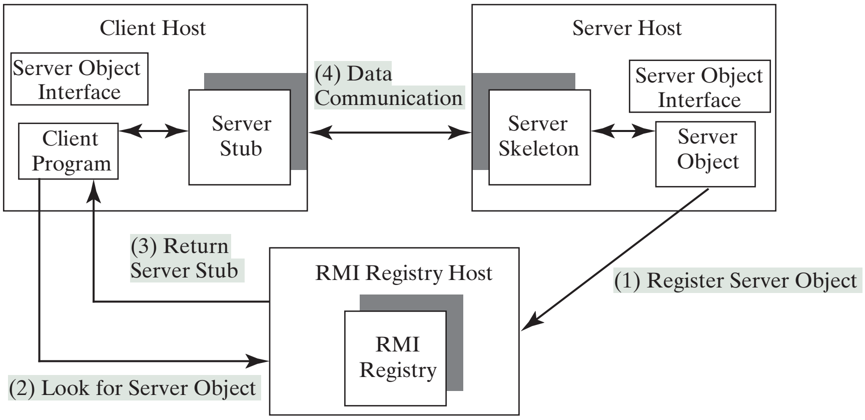


FIGURE 40.1 Java RMI uses a registry to provide naming services for remote objects, and uses the stub and the skeleton to facilitate communications between client and server.

RMI works as follows:

1. A server object is registered with the RMI registry.
2. A client looks through the RMI registry for the remote object.
3. Once the remote object is located, its stub is returned in the client.
4. The remote object can be used in the same way as a local object. Communication between the client and the server is handled through the stub and the skeleton.

The implementation of the RMI architecture is complex, but the good news is that RMI provides a mechanism that liberates you from writing the tedious code for handling parameter passing and invoking remote methods. The basic idea is to use two helper classes known as the *stub* and the *skeleton* for handling communications between client and server.

The stub and the skeleton are automatically generated. The *stub* resides on the client machine. It contains all the reference information the client needs to know about the server object. When a client invokes a method on a server object, it actually invokes a method that is encapsulated in the stub. The stub is responsible for sending parameters to the server and for receiving the result from the server and returning it to the client.

The *skeleton* communicates with the stub on the server side. The skeleton receives parameters from the client, passes them to the server for execution, and returns the result to the stub.

40.2.2 Passing Parameters

When a client invokes a remote method with parameters, passing the parameters is handled by the stub and the skeleton. Obviously, invoking methods in a remote object on a server is very different from invoking methods in a local object on a client, since the remote object is in a different address space on a separate machine. Let us consider three types of parameters:

- **Primitive data types**, such as `char`, `int`, `double`, or `boolean`, are passed by value like a local call.
- **Local object types**, such as `java.lang.String`, are also passed by value, but this is completely different from passing an object parameter in a local call. In a local call, an object parameter's reference is passed, which corresponds to the memory address of the object. In a remote call, there is no way to pass the object reference, because the address on one machine is meaningless to a different JVM. Any object can be used as

a parameter in a remote call as long as it is serializable. The stub serializes the object parameter and sends it in a stream across the network. The skeleton deserializes the stream into an object.

- **Remote object types** are passed differently from local objects. When a client invokes a remote method with a parameter of a remote object type, the stub of the remote object is passed. The server receives the stub and manipulates the parameter through it. Passing remote objects will be discussed in Section 40.6, "RMI Callbacks."

40.2.3 RMI Registry

How does a client locate the remote object? The RMI registry provides the registry services for the server to register the object and for the client to locate the object.

You can use several overloaded static `getRegistry()` methods in the `LocateRegistry` class to return a reference to a `Registry`, as shown in Figure 40.2. Once a `Registry` is obtained, you can bind an object with a unique name in the registry using the `bind` or `rebind` method or locate an object using the lookup method, as shown in Figure 40.3.

java.rmi.registry.LocateRegistry	
`+getRegistry(): Registry`	Returns a reference to the remote object Registry for the local host on the default registry port of 1099.
`+getRegistry(port: int): Registry`	Returns a reference to the remote object Registry for the local host on the specified port.
`+getRegistry(host: String): Registry`	Returns a reference to the remote object Registry on the specified host on the default registry port of 1099.
`+getRegistry(host:String, port: int): Registry`	Returns a reference to the remote object Registry on the specified host and port.

FIGURE 40.2 The `LocateRegistry` class provides the methods for obtaining a registry on a host.

java.rmi.registry.Registry	
`+bind(name: String, obj: Remote): void`	Binds the specified name with the remote object.
`+rebind(name: String, obj: Remote): void`	Binds the specified name with the remote object. Any existing binding for the name is replaced.
`+unbind(name: String): void`	Destroys the binding for the specified name that is associated with a remote object.
`+list(name: String): String[]`	Returns an array of the names bound in the registry.
`+lookup(name: String): Remote`	Returns a reference, a stub, for the remote object associated with the specified name.

FIGURE 40.3 The `Registry` class provides the methods for binding and obtaining references to remote objects in a remote object registry.

40.3 Developing RMI Applications

Key Point

An RMI application consists of defining server object interface, defining a server object interface implementation class, creating and registering a server object, and developing a client program.

Now that you have a basic understanding of RMI, you are ready to write simple RMI applications. The steps in developing an RMI application are shown in Figure 40.4 and listed below.

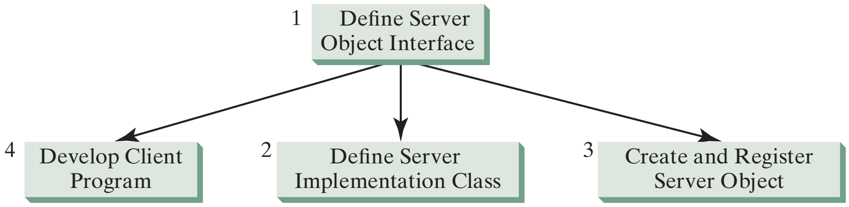

1 Define Server Object Interface
4 Develop Client Program
2 Define Server Implementation Class
3 Create and Register Server Object

FIGURE 40.4 The steps in developing an RMI application.

1. *Define a server object interface* that serves as the contract between the server and its clients, as shown in the following outline:

```
public interface ServerInterface extends Remote {
  public void service1(...) throws RemoteException;
  // Other methods
}
```

A server object interface must extend the `java.rmi.Remote` interface.

2. *Define a class that implements the server object interface*, as shown in the following outline:

```
public class ServerInterfaceImpl extends UnicastRemoteObject
    implements ServerInterface {
  public void service1(...) throws RemoteException {
    // Implement it
  }
  // Implement other methods
}
```

The server implementation class must extend the `java.rmi.server.UnicastRemoteObject` class. The `UnicastRemoteObject` class provides support for point-to-point active object references using TCP streams.

3. *Create a server object* from the server implementation class and register it with an RMI registry:

```
ServerInterface server = new ServerInterfaceImpl(...);
Registry registry = LocateRegistry.getRegistry();
registry.rebind("RemoteObjectName", server);
```

4. *Develop a client* that locates a remote object and invokes its methods, as shown in the following outline:

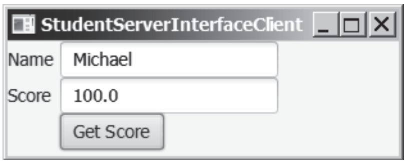
```
Registry registry = LocateRegistry.getRegistry(host);
ServerInterface server = (ServerInterfaceImpl)
  registry.lookup("RemoteObjectName");
server.service1(...);
```

The example that follows demonstrates the development of an RMI application through these steps.

40.3.1 Example: Retrieving Student Scores from an RMI Server

This example creates a client that retrieves student scores from an RMI server. The client, shown in Figure 40.5, displays the score for the specified name.

1. Create a server interface named `StudentServerInterface` in Listing 40.1. The interface tells the client how to invoke the server's `findScore` method to retrieve a student score.



Name Michael

Score 100.0

FIGURE 40.5 You can get the score by entering a student name and clicking the Get Score button.

LISTING 40.1 `StudentServerInterface.java`

```java
 1  import java.rmi.*;
 2
 3  public interface StudentServerInterface extends Remote {
 4    /**
 5     * Return the score for the specified name
 6     * @param name the student name
 7     * @return a double score or –1 if the student is not found
 8     */
 9    public double findScore(String name) throws RemoteException;
10  }
```

Any object that can be used remotely must be defined in an interface that extends the `java.rmi.Remote` interface (line 3). `StudentServerInterface`, extending `Remote`, defines the `findScore` method that can be remotely invoked by a client to find a student's score. Each method in this interface must declare that it may throw a `java.rmi.RemoteException` (line 9). Therefore your client code that invokes this method must be prepared to catch this exception in a try-catch block.

2. Create a server implementation named `StudentServerInterfaceImpl` (Listing 40.2) that implements `StudentServerInterface`. The `findScore` method returns the score for a specified student. It returns `-1` if the score is not found.

LISTING 40.2 `StudentServerInterfaceImpl.java`

```java
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class StudentServerInterfaceImpl
  extends UnicastRemoteObject
  implements StudentServerInterface {
  // Stores scores in a map indexed by name
  private HashMap<String, Double> scores =
    new HashMap<String, Double>();

  public StudentServerInterfaceImpl() throws RemoteException {
    initializeStudent();
  }

  /** Initialize student information */
  protected void initializeStudent() {
    scores.put("John", new Double(90.5));
    scores.put("Michael", new Double(100));
    scores.put("Michelle", new Double(98.5));
  }

  /** Implement the findScore method from the
   * Student interface */
  public double findScore(String name) throws RemoteException {
    Double d = (Double)scores.get(name);
27
28      if (d == null) {
29        System.out.println("Student " + name + " is not found ");
30        return -1;
31      }
32      else {
33        System.out.println("Student " + name + "\'s score is "
34          + d.doubleValue());
35        return d.doubleValue();
36      }
37    }
38  }
```

The `StudentServerInterfaceImpl` class implements `StudentServerInterface`. This class must also extend the `java.rmi.server.RemoteServer` class or its subclass. `RemoteServer` is an abstract class that defines the methods needed to create and export remote objects. Often its subclass `java.rmi.server.UnicastRemoteObject` is used (line 6). This subclass implements all the abstract methods defined in `RemoteServer`.

`StudentServerInterfaceImpl` implements the `findScore` method (lines 25–37) defined in `StudentServerInterface`. For simplicity, three students, John, Michael, and Michelle, and their corresponding scores are stored in an instance of `java.util.HashMap` named `scores`. `HashMap` is a concrete class of the `Map` interface in the Java Collections Framework, which makes it possible to search and retrieve a value using a key. Both values and keys are of `Object` type. The `findScore` method returns the score if the name is in the hash map, and returns `-1` if the name is not found.

3. Create a server object from the server implementation and register it with the RMI server (Listing 40.3).

LISTING 40.3 `RegisterWithRMIServer.java`

```
 1  import java.rmi.registry.*;
 2
 3  public class RegisterWithRMIServer {
 4    /** Main method */
 5    public static void main(String[] args) {
 6      try {
 7        StudentServerInterface obj =
 8          new StudentServerInterfaceImpl();
 9        Registry registry = LocateRegistry.getRegistry();
10        registry.rebind("StudentServerInterfaceImpl", obj);
11        System.out.println("Student server " + obj + " registered");
12      }
13      catch (Exception ex) {
14        ex.printStackTrace();
15      }
16    }
17  }
```

`RegisterWithRMIServer` contains a main method, which is responsible for starting the server. It performs the following tasks: (1) create a server object (line 8); (2) obtain a reference to the RMI registry (line 9), and (3) register the object in the registry (line 10).

4. Create a client named `StudentServerInterfaceClient` in Listing 40.4. The client locates the server object from the RMI registry and uses it to find the scores.

LISTING 40.4 `StudentServerInterfaceClient.java`

```
 1  import javafx.application.Application;
 2  import javafx.scene.Scene;
```

```
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class StudentServerInterfaceClient extends Application {
  // Declare a Student instance
  private StudentServerInterface student;

  private Button btGetScore = new Button("Get Score");
  private TextField tfName = new TextField();
  private TextField tfScore = new TextField();

  public void start(Stage primaryStage) {
    GridPane gridPane = new GridPane();
    gridPane.setHgap(5);
    gridPane.add(new Label("Name"), 0, 0);
    gridPane.add(new Label("Score"), 0, 1);
    gridPane.add(tfName, 1, 0);
    gridPane.add(tfScore, 1, 1);
    gridPane.add(btGetScore, 1, 2);

    // Create a scene and place the pane in the stage
    Scene scene = new Scene(gridPane, 250, 250);
    primaryStage.setTitle("StudentServerInterfaceClient");
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    initializeRMI();
    btGetScore.setOnAction(e - > getScore());
  }

  private void getScore() {
    try {
      // Get student score
      double score = student.findScore(tfName.getText().trim());

      // Display the result
      if (score < 0)
        tfScore.setText("Not found");
      else
        tfScore.setText(new Double(score).toString());
    }
    catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  /** Initialize RMI */
  protected void initializeRMI() {
    String host = "";

    try {
      Registry registry = LocateRegistry.getRegistry(host);
      student = (StudentServerInterface)
        registry.lookup("StudentServerInterfaceImpl");
      System.out.println("Server object " + student + " found");
    }
    catch(Exception ex) {
```

```
65        System.out.println(ex);
66      }
67    }
68
69    /**
70     * The main method is only needed for the IDE with limited
71     * JavaFX support. Not needed for running from the command line.
72     */
73    public static void main(String[] args) {
74      launch(args);
75    }
76  }
```

`StudentServerInterfaceClient` invokes the `findScore` method on the server to find the score for a specified student. The key method in `StudentServerInterfaceClient` is the `initializeRMI` method (lines 55–67), which is responsible for locating the server stub.

The `lookup(String name)` method (line 61) returns the remote object with the specified name. Once a remote object is found, it can be used just like a local object. The stub and the skeleton are used behind the scenes to make the remote method invocation work.

5. Follow the steps below to run this example.

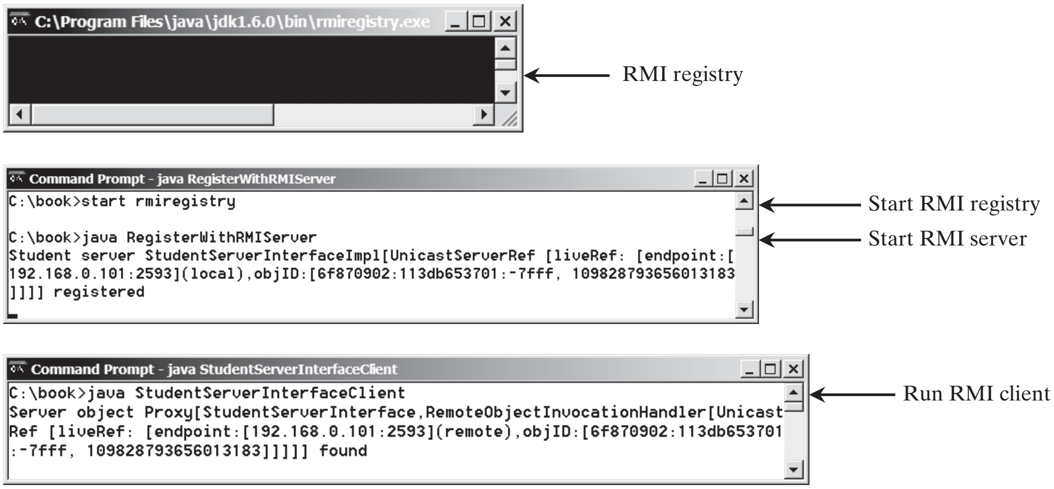
   5.1. Start the RMI registry by typing "**start rmiregistry**" at a DOS prompt from the book directory. By default, the port number `1099` is used by rmiregistry. To use a different port number, simply type the command "**start rmiregistry *portnumber***" at a DOS prompt.

   5.2. Start the server `RegisterWithRMIServer` using the following command at C:\book directory:

   C:\ book>**java RegisterWithRMIServer**

   5.3. Run the client `StudentServerInterfaceClient` as an application. A sample run of the application is shown in Figure 40.5(b).

Note:
You must start rmiregistry from the directory where you will run the RMI server, as shown in Figure 40.6. Otherwise, you will receive the error `ClassNotFoundException` on `StudentServerInterfaceImpl_Stub`.


C:\Program Files\java\jdk1.6.0\bin\rmiregistry.exe
Command Prompt - java RegisterWithRMIServer
Command Prompt - java StudentServerInterfaceClient

```
C:\book>start rmiregistry

C:\book>java RegisterWithRMIServer
Student server StudentServerInterfaceImpl[UnicastServerRef [liveRef: [endpoint:[
192.168.0.101:2593](local),objID:[6f870902:113db653701:-7fff, 109828793656013183
]]]] registered
```

```
C:\book>java StudentServerInterfaceClient
Server object Proxy[StudentServerInterface,RemoteObjectInvocationHandler[Unicast
Ref [liveRef: [endpoint:[192.168.0.101:2593](remote),objID:[6f870902:113db653701
:-7fff, 109828793656013183]]]]] found
```

RMI registry

Start RMI registry

Start RMI server

Run RMI client

FIGURE 40.6 To run an RMI program, first start the RMIRegistry, then register the server object with the registry. The client locates it from the registry.

Note:
Server, registry, and client can be on three different machines. If you run the client and the server on separate machines, you need to place `StudentServerInterface` on both machines.

Caution:
If you modify the remote object implementation class, you need to restart the server class to reload the object to the RMI registry. In some old versions of rmiregistry, you may have to restart rmiregistry.

Check Point

40.3.1 How do you define an interface for a remote object?

40.3.2 Describe the roles of the stub and the skeleton.

40.3.3 What is `java.rmi.Remote`? How do you define a server class?

40.3.4 What is an RMI registry for? How do you create an RMI registry?

40.3.5 What is the command to start an RMI registry?

40.3.6 How do you register a remote object with the RMI registry?

40.3.7 What is the command to start a custom RMI server?

40.3.8 How does a client locate a remote object stub through an RMI registry?

40.3.9 How do you obtain a registry? How do you register a remote object? How do you locate remote object?

40.4 RMI vs. Socket-Level Programming

Key Point

RMI is a high-level network programming and socket-level network programming is low-low-level.

RMI enables you to program at a higher level of abstraction. It hides the details of socket server, socket, connection, and sending or receiving data. It even implements a multithreading server under the hood, whereas with socket-level programming, you have to explicitly implement threads for handling multiple clients.

RMI applications are scalable and easy to maintain. You can change the RMI server or move it to another machine without modifying the client program except for resetting the URL to locate the server. (To avoid resetting the URL, you can modify the client to pass the URL as a command-line parameter.) In socket-level programming, a client operation to send data requires a server operation to read it. The implementation of client and server at the socket level is tightly synchronized.

RMI clients can directly invoke the server method, whereas socket-level programming is limited to passing values. Socket-level programming is very primitive. Avoid using it to develop client/server applications. As an analogy, socket-level programming is like programming in assembly language, whereas RMI programming is like programming in a high-level language.

Check Point

40.10 What are the advantages of RMI over socket-level programming?

40.5 Developing Three-Tier Applications Using RMI

Key Point

RMI can be used in the middle between a client and a database to develop scalable and flexible business applications.

Three-tier applications have gained considerable attention in recent years, largely because of the demand for more scalable and load-balanced systems to replace traditional two-tier client/server database systems. A centralized database system does not just handle data access, but it also processes the business rules on data. Thus, a centralized database is usually heavily loaded, because it requires extensive data manipulation and processing. In some situations, data processing is handled by the client and business rules are stored on the client side. It is preferable to use a middle tier as a buffer between client and database. The middle tier can be used to apply business logic and rules, and to process data to reduce the load on the database.

A three-tier architecture does more than just reduce the processing load on the server. It also provides access to multiple network sites. This is especially useful to Java clients that need to access multiple databases on different servers, since the server may change.

To demonstrate, let us rewrite the example in Section 40.3.1, "Example: Retrieving Student Scores from an RMI Server," to find scores stored in a database rather than a hash map. In addition, the system is capable of blocking a client from accessing a student who has not given the university permission to publish his/her score. An RMI component is developed to serve as a middle tier between client and database; it sends a search request to the database, processes the result, and returns an appropriate value to the client.

For simplicity, this example reuses the `StudentServerInterface` interface and `StudentServerInterfaceClient` class from Section 40.3.1 with no modifications. All you have to do is to provide a new implementation for the server interface and create a program to register the server with the RMI. Here are the steps to complete the program:

1. Store the scores in a database table named `Score` that contains three columns: `name`, `score`, and `permission`. The permission value is `1` or `0`, which indicates whether the student has given the university permission to release his/her grade. The following is the statement to create the table and insert three records:

```
create table Scores (name varchar(20),
  score number, permission number);

insert into Scores values ('John', 90.5, 1);
insert into Scores values ('Michael', 100, 1);
insert into Scores values ('Michelle', 100, 0);
```

2. Create a new server implementation named `Student3TierImpl` in Listing 40.5. The server retrieves a record from the `Scores` table, processes the retrieved information, and sends the result back to the client.

LISTING 40.5 Student3TierImpl.java

```java
import java.rmi.*;
import java.rmi.server.*;
import java.sql.*;

public class Student3TierImpl extends UnicastRemoteObject
    implements StudentServerInterface {
  // Use prepared statement for querying DB
  private PreparedStatement pstmt;

  /** Constructs Student3TierImpl object and exports it on
  * default port.
  */
  public Student3TierImpl() throws RemoteException {
    initializeDB();
  }

  /** Constructs Student3TierImpl object and exports it on
   * specified port.
   * @param port The port for exporting
   */
  public Student3TierImpl(int port) throws RemoteException {
    super(port);
    initializeDB();
  }

  /** Load JDBC driver, establish connection and
   * create statement */
  protected void initializeDB() {
    try {
      // Load the JDBC driver
      // Class.forName("oracle.jdbc.driver.OracleDriver");
      Class.forName("com.mysql.jdbc.Driver ");

      System.out.println("Driver registered");

      // Establish connection
      /*Connection conn = DriverManager.getConnection
        ("jdbc:oracle:thin:@drake.armstrong.edu:1521:orcl",
          "scott", "tiger"); */
      Connection conn = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook", "scott", "tiger");
      System.out.println("Database connected");

      // Create a prepared statement for querying DB
      pstmt = conn.prepareStatement(
        "select * from Scores where name = ?");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }

  /** Return the score for specified the name
   * Return -1 if score is not found.
   */
  public double findScore(String name) throws RemoteException {
    double score = -1;
    try {
      // Set the specified name in the prepared statement
      pstmt.setString(1, name);

      // Execute the prepared statement
      ResultSet rs = pstmt.executeQuery();

      // Retrieve the score
      if (rs.next()) {
        if (rs.getBoolean(3))
          score = rs.getDouble(2);
      }
    }
    catch (SQLException ex) {
      System.out.println(ex);
    }

    return score;
  }
}
```

`Student3TierImpl` is similar to StudentServerInterfaceImpl in Section 40.3.1 except that the `Student3TierImpl` class finds the score from a JDBC data source instead from a hash map.

The table named `Scores` consists of three columns, `name`, `score`, and `permission`, where the latter indicates whether the student has given permission to show his/her score. Since SQL does not support a `boolean` type, permission is defined as a number whose value of `1` indicates `true` and of `0` indicates `false`.

The `initializeDB()` method (lines 28–51) establishes connections with the database and creates a prepared statement for processing the query.

The `findScore` method (lines 56–76) sets the name in the prepared statement, executes the statement, processes the result, and returns the score for a student whose permission is `true`.

3. Write a `main` method in the class `RegisterStudent3TierServer` (Listing 40.6) that registers the server object using StudentServerInterfaceImpl, the same name as in Listing 40.2, so that you can use `StudentServerInterfaceClient`, created in Section 40.3.1, to test the server.

LISTING 40.6 `RegisterStudent3TierServer.java`

```java
import java.rmi.registry.*;

public class RegisterStudent3TierServer {
  public static void main(String[] args) {
    try {
      StudentServerInterface obj = new Student3TierImpl();
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind("StudentServerInterfaceImpl", obj);
      System.out.println("Student server " + obj + " registered");
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

4. Follow the steps below to run this example.

   4.1. Start RMI registry by typing "**start rmiregistry**" at a DOS prompt from the book directory.

   4.2. Start the server `RegisterStudent3TierServer` using the following command at the C:\ book directory:

   C:\ book>`java RegisterStudent3TierServer`

   4.3. Run the client `StudentServerInterfaceClient`. A sample run is shown in Figure 40.6.

40.5.1 Describe how parameters are passed in RMI.

Check Point

40.6 RMI Callbacks

RMI callbacks enable the server to invoke the methods on a client.

Key Point

In a traditional client/server system, a client sends a request to a server, and the server processes the request and returns the result to the client. The server cannot invoke the methods on a client. One important benefit of RMI is that it supports *callbacks*, which enable the server to invoke methods on the client. With the RMI callback feature, you can develop interactive distributed applications.

In Section 33.6, "Case Studies: Distributed TicTacToe Games," you developed a distributed TicTacToe game using stream socket programming. The example that follows demonstrates the use of the RMI callback feature to develop an interactive TicTacToe game.

All the examples you have seen so far in this chapter have simple behaviors that are easy to model with classes. The behavior of the TicTacToe game is somewhat complex. To create the classes to model the game, you need to study and understand it and distribute the process appropriately between client and server.

Clearly the client should be responsible for handling user interactions, and the server should coordinate with the client. Specifically, the client should register with the server, and the server can take two and only two players. Once a client makes a move, it should notify the server; the server then notifies the move to the other player. The server should determine the status of the game—that is, whether it has been won or drawn—and notify the players. The server should also coordinate the turns—that is, which client has the turn at a given time. The ideal approach for notifying a player is to invoke a method in the client that sets appropriate properties in the client or sends messages to a player. Figure 40.7 illustrates the relationship between clients and server.

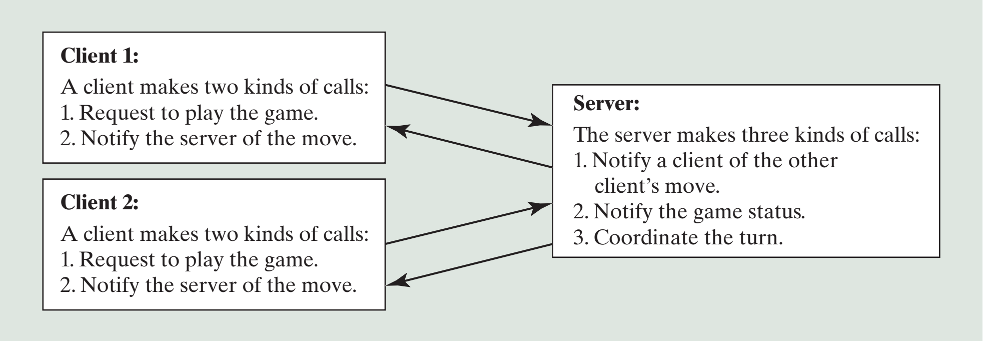

Client 1:
A client makes two kinds of calls:
1. Request to play the game.
2. Notify the server of the move.
Client 2:
A client makes two kinds of calls:
1. Request to play the game.
2. Notify the server of the move.
Server:
The server makes three kinds of calls:
1. Notify a client of the other client's move.
2. Notify the game status.
3. Coordinate the turn.

FIGURE 40.7 The server coordinates the activities with the clients.

All the calls a client makes can be encapsulated in one remote interface named `TicTacToe` (Listing 40.7), and all the calls the server invokes can be defined in another interface named `CallBack` (Listing 40.8). These two interfaces are defined as follows:

LISTING 40.7 TicTacToeInterface.java

```
import java.rmi.*;

public interface TicTacToeInterface extends Remote {
  /**
   * Connect to the TicTacToe server and return the token.
   * If the returned token is ' ', the client is not connected to
   * the server
   */
  public char connect(CallBack client) throws RemoteException;

  /** A client invokes this method to notify the server of its move*/
  public void myMove(int row, int column, char token)
    throws RemoteException;
}
```

LISTING 40.8 CallBack.java

```java
import java.rmi.*;

public interface CallBack extends Remote {
  /** The server notifies the client for taking a turn */
  public void takeTurn(boolean turn) throws RemoteException;

  /** The server sends a message to be displayed by the client */
  public void notify(java.lang.String message)
    throws RemoteException;

  /** The server notifies a client of the other player's move */
  public void mark(int row, int column, char token)
    throws RemoteException;
}
```

What does a client need to do? The client interacts with the player. Assume that all the cells are initially empty, and that the first player takes the X token and the second player the O token. To mark a cell, the player points the mouse to the cell and clicks it. If the cell is empty, the token (X or O) is displayed. If the cell is already filled, the player's action is ignored.

From the preceding description, it is obvious that a cell is a GUI object that handles mouse-click events and displays tokens. The candidate for such an object could be a button or a panel. Panels are more flexible than buttons. The token (X or O) can be drawn on a panel in any size, but it can be displayed only as a label on a button.

Let `Cell` be a subclass of `JPanel`. You can declare a 3 × 3 grid to be an array `Cell[][] cell = new Cell[3][3]` for modeling the game. How do you know the state of a cell (marked or not)? You can use a property named `marked` of the `boolean` type in the `Cell` class. How do you know whether the player has a turn? You can use a property named `myTurn` of `boolean`. This property (initially `false`) can be set by the server through a callback.

The `Cell` class is responsible for drawing the token when an empty cell is clicked, so you need to write the code for listening to the `MouseEvent` and for painting the shape for tokens X and O. To determine which shape to draw, introduce a variable named `marker` of the `char` type. Since this variable is shared by all the cells in a client, it is preferable to declare it in the client and to declare the `Cell` class as an inner class of the client so that this variable will be accessible to all the cells.

Now let us turn our attention to the server side. What does the server need to do? The server needs to implement `TicTacToeInterface` and notify the clients of the game status. The server has to record the moves in the cells and check the status every time a player makes a move. The status information can be kept in a 3 × 3 array of `char`. You can implement a method named `isFull()` to check whether the board is full and a method named `isWon(token)` to check whether a specific player has won.

Once a client is connected to the server, the server notifies the client which token to use—that is, X for the first client and O for the second. Once a client notifies the server of its move, the server checks the game status and notifies the clients.

Now the most critical question is how the server notifies a client. You know that a client invokes a server method by creating a server stub on the client side. A server cannot directly invoke a client, because the client is not declared as a remote object. The `CallBack` interface was created to facilitate the server's callback to the client. In the implementation of `CallBack`, an instance of the client is passed as a parameter in the constructor of `CallBack`. The client creates an instance of `CallBack` and passes its stub to the server, using a remote method named `connect()` defined in the server. The server then invokes the client's method through a `CallBack` instance. The triangular relationship of client, `CallBack` implementation, and server is shown in Figure 40.8.

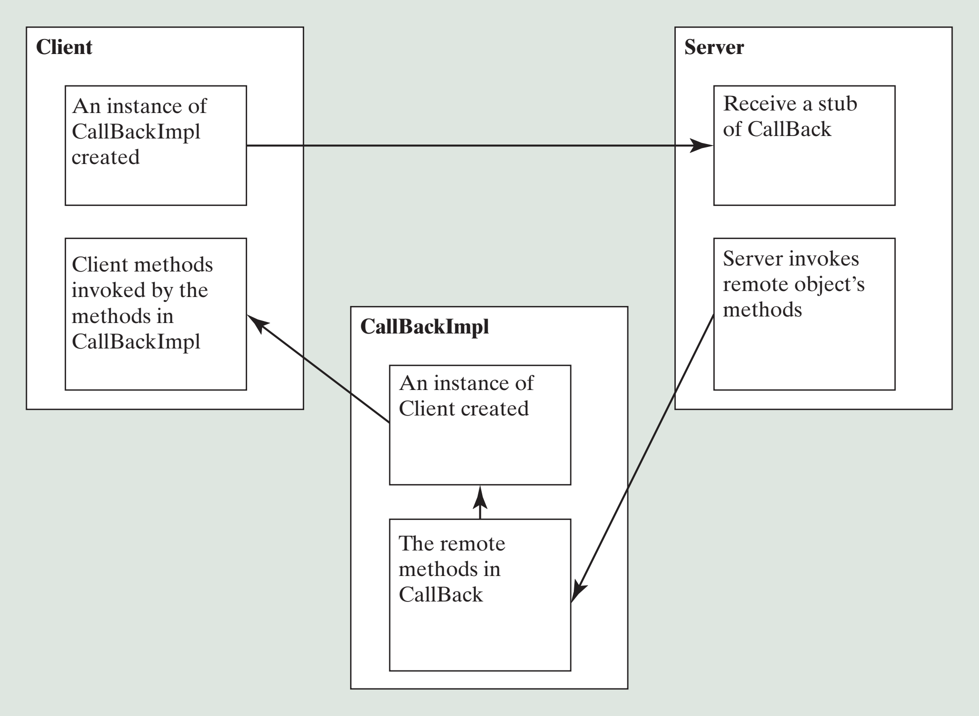

Client
An instance of CallBackImpl created
Client methods invoked by the methods in CallBackImpl
Server
Receive a stub of CallBack
Server invokes remote object's methods
CallBackImpl
An instance of Client created
The remote methods in CallBack

FIGURE 40.8 The server receives a `CallBack` stub from the client and invokes the remote methods defined in the `CallBack` interface, which can invoke the methods defined in the client.

Here are the steps to complete the example.

1. Create TicTacToeImpl.java (Listing 40.9) to implement `TicTacToeInterface`. Add a main method in the program to register the server with the RMI.

LISTING 40.9 `TicTacToeImpl.java`

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.rmi.registry.*;

public class TicTacToeImpl extends UnicastRemoteObject
    implements TicTacToeInterface {
  // Declare two players, used to call players back
  private CallBack player1 = null;
  private CallBack player2 = null;

  // board records players' moves
  private char[][] board = new char[3][3];

  /** Constructs TicTacToeImpl object and
    exports it on default port.
   */
  public TicTacToeImpl() throws RemoteException {
    super();
  }

  /** Constructs TicTacToeImpl object and exports it on specified
   * port.
   * @param port The port for exporting
   */
  public TicTacToeImpl(int port) throws RemoteException {
    super(port);
  }

  /**
   * Connect to the TicTacToe server and return the token.
   * If the returned token is ' ', the client is not connected to
   * the server
   */
  public char connect(CallBack client) throws RemoteException {
    if (player1 == null) {
      // player1 (first player) registered
      player1 = client;
      player1.notify("Wait for a second player to join");
      return 'X';
    }
    else if (player2 == null) {
      // player2 (second player) registered
      player2 = client;
      player2.notify("Wait for the first player to move");
      player2.takeTurn(false);
      player1.notify("It is my turn (X token)");
      player1.takeTurn(true);
      return 'O';
    }
    else {
      // Already two players
      client.notify("Two players are already in the game");
      return ' ';
    }
  }

  /** A client invokes this method to notify the
    server of its move*/
  public void myMove(int row, int column, char token)
      throws RemoteException {
    // Set token to the specified cell
    board[row][column] = token;

    // Notify the other player of the move
    if (token == 'X')
      player2.mark(row, column, 'X');
    else
      player1.mark(row, column, 'O');

    // Check if the player with this token wins
    if (isWon(token)) {
      if (token == 'X') {
        player1.notify("I won!");
        player2.notify("I lost!");
        player1.takeTurn(false);
      }
      else {
        player2.notify("I won!");
        player1.notify("I lost!");
        player2.takeTurn(false);
      }
    }
    else if (isFull()) {
      player1.notify("Draw!");
      player2.notify("Draw!");
      }
      else if (token == 'X') {
        player1.notify("Wait for the second player to move");
        player1.takeTurn(false);
        player2.notify("It is my turn, (O token)");
        player2.takeTurn(true);
      }
      else if (token == 'O') {
        player2.notify("Wait for the first player to move");
        player2.takeTurn(false);
        player1.notify("It is my turn, (X token)");
        player1.takeTurn(true);
      }
    }

    /** Check if a player with the specified token wins */
    public boolean isWon(char token) {
      for (int i = 0; i < 3; i++)
        if ((board[i][0] == token) && (board[i][1] == token)
          && (board[i][2] == token))
          return true;

      for (int j = 0; j < 3; j++)
        if ((board[0][j] == token) && (board[1][j] == token)
          && (board[2][j] == token))
          return true;

      if ((board[0][0] == token) && (board[1][1] == token)
        && (board[2][2] == token))
        return true;

      if ((board[0][2] == token) && (board[1][1] == token)
        && (board[2][0] == token))
        return true;

      return false;
    }

    /** Check if the board is full */
    public boolean isFull() {
      for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
          if (board[i][j] == '\u0000')
            return false;

      return true;
    }

    public static void main(String[] args) {
      try {
        TicTacToeInterface obj = new TicTacToeImpl();
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind("TicTacToeImpl", obj);
        System.out.println("Server " + obj + " registered");
      }
      catch (Exception ex) {
        ex.printStackTrace();
      }
    }
  }
```

2. Create CallBackImpl.java (Listing 40.10) to implement the `CallBack` interface.

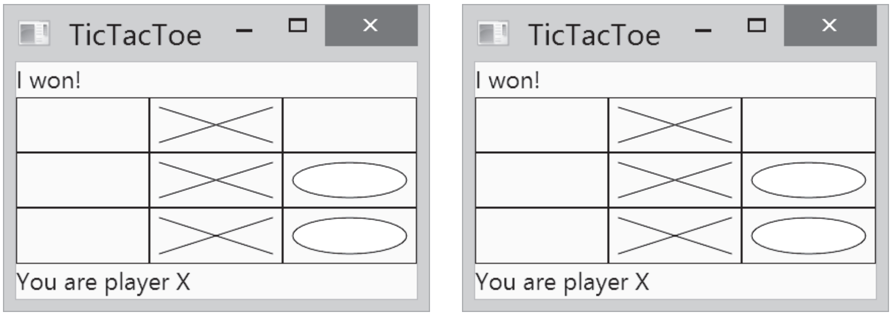
LISTING 40.10 `CallBackImpl.java`

```java
import java.rmi.*;
import java.rmi.server.*;

public class CallBackImpl extends UnicastRemoteObject
    implements CallBack {
  // The client will be called by the server through callback
  private TicTacToeClientRMI thisClient;

  /** Constructor */
  public CallBackImpl(Object client) throws RemoteException {
    thisClient = (TicTacToeClientRMI)client;
  }

  /** The server notifies the client for taking a turn */
  public void takeTurn(boolean turn) throws RemoteException {
    thisClient.setMyTurn(turn);
  }

  /** The server sends a message to be displayed by the client */
  public void notify(String message )throws RemoteException {
    thisClient.setMessage(message);
  }

  /** The server notifies a client of the other player's move */
  public void mark(int row, int column, char token)
      throws RemoteException {
    thisClient.mark(row, column, token);
  }
}
```

3. Create a client named `TicTacToeClientRMI` (Listing 40.11) for interacting with a player and communicating with the server. Enable it to run standalone.

LISTING 40.11 `TicTacToeClientRMI.java`

```java
import java.rmi.*;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.scene.shape.Ellipse;

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;

public class TicTacToeClientRMI extends Application {
  // marker is used to indicate the token type
  private char marker;

  // myTurn indicates whether the player can move now
  private boolean myTurn = false;

  // Indicate which player has a turn, initially it is the X player
  private char whoseTurn = 'X';

  // Create and initialize cell
  private Cell[][] cell = new Cell[3][3];

  // Create and initialize a status label
  private Label lblStatus = new Label("X's turn to play");

  // ticTacToe is the game server for coordinating
  // with the players
  private TicTacToeInterface ticTacToe;

  private Label lblIdentification = new Label();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Pane to hold cell
    GridPane pane = new GridPane();
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        pane.add(cell[i][j] = new Cell(i, j), j, i);

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(pane);
    borderPane.setTop(lblStatus);
    borderPane.setBottom(lblIdentification);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 450, 170);
    primaryStage.setTitle("TicTacToe"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread( () -> {
    try {
      initializeRMI();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }}).start();
  }

  /** Initialize RMI */
  protected boolean initializeRMI() throws Exception {
    String host = "";

    try {
      Registry registry = LocateRegistry.getRegistry(host);
      ticTacToe = (TicTacToeInterface)
        registry.lookup("TicTacToeImpl");
      System.out.println
        ("Server object " + ticTacToe + " found");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }

    // Create callback for use by the
    // server to control the client
    CallBackImpl callBackControl = new CallBackImpl(this);

    if (
      (marker =
        ticTacToe.connect((CallBack)callBackControl)) != ' ')
    {
      System.out.println("connected as " + marker + " player.");
      Platform.runLater(() ->
        lblIdentification.setText("You are player " + marker));
      return true;
    }
    else {
      System.out.println("already two players connected as ");
      return false;
    }
  }

  /** Set variable myTurn to true or false */
  public void setMyTurn(boolean myTurn) {
    this.myTurn = myTurn;
  }

  /** Set message on the status label */
  public void setMessage(String message) {
    Platform.runLater(() -> lblStatus.setText(message));
  }

  /** Mark the specified cell using the token */
  public void mark(int row, int column, char token) {
    cell[row][column].setToken(token);
  }

  // An inner class for a cell
  public class Cell extends Pane {
    // marked indicates whether the cell has been used
    private boolean marked = false;

    // row and column indicate where the cell appears on the board
    int row, column;

    // Token used for this cell
    private char token = ' ';

    public Cell(final int row, final int column) {
      this.row = row;
      this.column = column;
      setStyle("-fx-border-color: black");
      this.setPrefSize(2000, 2000);
      this.setOnMouseClicked(e -> handleMouseClick());
    }

    /** Return token */
    public char getToken() {
      return token;
    }

    /** Set a new token */
    public void setToken(char c) {
      token = c;
      marked = true;

      if (token == 'X') {
        Line line1 = new Line(10, 10,
          this.getWidth() - 10, this.getHeight() - 10);
        line1.endXProperty().bind(this.widthProperty().subtract(10));
        line1.endYProperty().bind(this.heightProperty().subtract(10));
        Line line2 = new Line(10, this.getHeight() - 10,
          this.getWidth() - 10, 10);
        line2.startYProperty().bind(
          this.heightProperty().subtract(10));
        line2.endXProperty().bind(this.widthProperty().subtract(10));

        // Add the lines to the pane
        Platform.runLater(() ->
          this.getChildren().addAll(line1, line2));
      }
      else if (token == 'O') {
        Ellipse ellipse = new Ellipse(this.getWidth() / 2,
          this.getHeight() / 2, this.getWidth() / 2 - 10,
          this.getHeight() / 2 - 10);
        ellipse.centerXProperty().bind(
          this.widthProperty().divide(2));
        ellipse.centerYProperty().bind(
          this.heightProperty().divide(2));
        ellipse.radiusXProperty().bind(
          this.widthProperty().divide(2).subtract(10));
        ellipse.radiusYProperty().bind(
          this.heightProperty().divide(2).subtract(10));
        ellipse.setStroke(Color.BLACK);
        ellipse.setFill(Color.WHITE);

        Platform.runLater(() ->
          getChildren().add(ellipse)); // Add the ellipse to the pane
      }
    }

    /* Handle a mouse click event */
    private void handleMouseClick() {
      if (myTurn && !marked) {
        // Mark the cell
        setToken(marker);

        // Notify the server of the move
        try {
          ticTacToe.myMove(row, column, marker);
        }
        catch (RemoteException ex) {
          System.out.println(ex);
        }
      }
    }
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```



I won!

You are player X

I won!

You are player X

Figure 40.9 Two players play each other through the RMI server.

4. Follow the steps below to run this example.
    - 4.1. Start RMI registry by typing "**start rmiregistry**" at a DOS prompt from the book directory.
    - 4.2. Start the server `TicTacToeImpl` using the following command at the C:\ book directory:

      ```
      C:\ book>java TicTacToeImpl
      ```
    - 4.3. Run the client `TicTacToeClientRMI`. A sample run is shown in Figure 40.9.

`TicTacToeInterface` defines two remote methods, `connect(CallBack client)` and `myMove(int row, int column, char token)`. The `connect` method plays two roles: one is to pass a `CallBack` stub to the server, and the other is to let the server assign a token for the player. The `myMove` method notifies the server that the player has made a specific move.

The `CallBack` interface defines three remote methods, `takeTurn(boolean turn)`, `notify(String message)`, and `mark(int row, int column, char token)`. The `takeTurn` method sets the client's `myTurn` property to `true` or `false`. The `notify` method displays a message on the client's status label. The `mark` method marks the client's cell with the token at the specified location.

`TicTacToeImpl` is a server implementation for coordinating with the clients and managing the game. The variables `player1` and `player2` are instances of `CallBack`, each of which corresponds to a client, passed from a client when the client invokes the `connect` method. The variable `board` records the moves by the two players. This information is needed to determine the game status. When a client invokes the `connect` method, the server assigns a token X for the first player and O for the second player, and accepts only two players. You can modify the program to accept additional clients as observers. See Exercise 40.7 for more details.

Once two players are in the game, the server coordinates the turns between them. When a client invokes the `myMove` method, the server records the move and notifies the other player by marking the other player's cell. It then checks to see whether the player wins or whether the board is full. If neither condition applies and therefore the game continues, the server gives a turn to the other player.

The `CallBackImpl` implements the `CallBack` interface. It creates an instance of `TicTacToeClientRMI` through its constructor. The `CallBackImpl` relays the server request to the client by invoking the client's methods. When the server invokes the `takeTurn` method, `CallBackImpl` invokes the client's `setMyTurn()` method to set the property `myTurn` in the client. When the server invokes the `notify()` method, `CallBackImpl` invokes the client's `setMessage()` method to set the message on the client's status label. When the server invokes the `mark` method, `CallBackImpl` invokes the client's `mark` method to mark the specified cell.

Interestingly, obtaining the `TicTacToeImpl` stub for the client is different from obtaining the `CallBack` stub for the server. The `TicTacToeImpl` stub is obtained by invoking the `lookup()` method through the RMI registry, and the `CallBack` stub is passed to the server through the `connect` method in the `TicTacToeImpl` stub. It is a common practice to obtain the first stub with the `lookup` method, but to pass the subsequent stubs as parameters through remote method invocations.

Since the variables `myTurn` and `marker` are defined in `TicTacToeClientRMI`, the `Cell` class is defined as an inner class within `TicTacToeClientRMI` in order to enable all the cells in the client to access them. Exercise 40.8 suggests alternative approaches that implement the `Cell` as a noninner class.

Check Point

40.6.1 What is the problem if the `connect` method in the `TicTacToeInterface` is defined as

```
public boolean connect(CallBack client, char token)
  throws RemoteException;
```

or as

```
public boolean connect(CallBack client, Character token)
  throws RemoteException;
```

40.6.2 What is callback? How does callback work in RMI?

KEY TERMS

callback 40-13	skeleton 40-3
RMI registry 40-3	stub 40-3

CHAPTER SUMMARY

1. RMI is a high-level Java API for building distributed applications using distributed objects.

2. The key idea of RMI is its use of stubs and skeletons to facilitate communications between objects. The stub and skeleton are automatically generated, which relieves programmers of tedious socket-level network programming.

3. For an object to be used remotely, it must be defined in an interface that extends the `java.rmi.Remote` interface.

4. In an RMI application, the initial remote object must be registered with the RMI registry on the server side and be obtained using the `lookup` method through the registry on the client side. Subsequent uses of stubs of other remote objects may be passed as parameters through remote method invocations.

5. RMI is especially useful for developing scalable and load-balanced multitier distributed applications.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

Section 40.3

***40.1** (*Limit the number of clients*) Modify the example in Section 40.3.1, "Example: Retrieving Student Scores from an RMI Server," to limit the number of concurrent clients to 10.

***40.2** (*Compute loan*) Rewrite Programming Exercise 33.1 using RMI. You need to define a remote interface for computing monthly payment and total payment.

****40.3** (*Web visit count*) Rewrite Programming Exercise 33.4 using RMI. You need to define a remote interface for obtaining and increasing the count.

****40.4** (*Display and add addresses*) Rewrite Programming Exercise 33.6 using RMI. You need to define a remote interface for adding addresses and retrieving address information.

Section 40.5

****40.5** (*Address in a database table*) Rewrite Programming Exercise 40.4. Assume that the address is stored in a table.

****40.6** (*Three-tier application*) Use the three-tier approach to modify Programming Exercise 40.4, as follows:

- Create a JavaFX client to manipulate student information, as shown in Figure 33.23a.
- Create a remote object interface with methods for retrieving, inserting, and updating student information, and an object implementation for the interface.

Section 40.6

****40.7** (*Chat*) Rewrite Programming Exercise 33.13 using RMI. You need to define a remote interface for sending and receiving a message.

****40.8** (*Improve TicTacToe*) Modify the TicTacToe example in Section 40.6, "RMI Callbacks," as follows:

- Allow a client to connect to the server as an observer to watch the game.
- Rewrite the `Cell` class as a noninner class.

CHAPTER 41

WEB SERVICES

Objectives

- To describe what a Web service is (§41.1).
- To create a Web service class (§41.2).
- To publish and test a Web service (§41.3).
- To create a Web service client reference (§41.4).
- To explain the role of WSDL (§41.4).
- To pass arguments of object type in a Web service (§41.5).
- To discover how a client communicates with a Web service (§41.5).
- To describe what SOAP requests and SOAP responses are (§41.5).
- To track a session in Web services (§41.6).

41.1 Introduction

Web services is about sharing objects on the Internet.

Web service is a technology that enables programs to communicate through HTTP on the Internet. Web services enable a program on one system to invoke a method in an object on another system. You can develop and use Web services using any languages on any platform. Web services are simple and easy to develop.

Web services run on the Web using HTTP. There are several APIs for Web services. A popular standard is the *Simple Object Access Protocol* (SOAP), which is based on XML. The computer on which a Web service resides is referred to as a *server*. The server needs to make the service available to the client, known as *publishing a Web service*. Using a Web service from a client is known as *consuming a Web service*.

A client interacts with a Web service through a *proxy object*. The proxy object facilitates the communication between the client and the Web service. The client passes arguments to invoke methods on the proxy object. The proxy object sends the request to the server and receives the result back from the server, as shown in Figure 41.1.

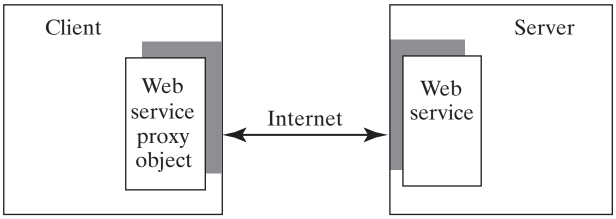


FIGURE 41.1 A proxy object serves as a facilitator between a client and a Web service.

41.1.1 What is a Web service?

41.1.2 Can you invoke a Web service from a language other than Java?

41.1.3 Do Web services support callback? That is, can a Web service call a method from a client's program?

41.1.4 What is SOAP? What is it to publish a Web service? What is it to consume a Web service? What is the role of a proxy object?

41.2 Creating Web Services

An IDE such as NetBeans is an effective tool for developing and deploying Web services.

There are many tools for creating Web services. This book demonstrates creating Web services using NetBeans.

> **Note**
> Apache Tomcat Server does not work well with Web services. To develop and deploy Web services using NetBeans, you need to install GlassFish. For information on how to install GlassFish on NetBeans.

We now create a Web service for obtaining student scores. A Web service is a class that contains the methods for the client to invoke. Name the class `ScoreService` with a method named `findScore(String name)` that returns the score for a student.

First, you need to create a *Web project* using the following steps:

1. Choose **`File, New Project`** to display the New Project dialog box. In the New Project dialog box, choose **`Java Web`** in the Categories pane and choose **`Web Application`** in the Projects pane. Click *Next* to display the New Web Application dialog box.

2. Enter `WebServiceProject` as the project name, specify the location where you want the project to be stored, and click Next to display the Server and Setting dialog.
3. Select **`GlassFish 4`** as the server and **`Java EE 7 Web`** as the Java EE version. Click *Finish* to create the project.

Now you can create the `ScoreService` class in the project as follows:

1. Right-click the `WebServiceProject` in the Project pane to display a context menu. Choose **`New, Web Service`** to display the New Web Service dialog box. (If you don't see **`Web Service`**, click **`New`**, **`Other`** to display the New File dialog box to choose Web Service in this dialog box.)
2. Enter `ScoreService` in the Web Service Name field and enter `chapter41` in the Package field. Click *Finish* to create `ScoreService`.
3. Complete the source code as shown in Listing 41.1.

LISTING 41.1 `ScoreService.java`

```
 1  package chapter41;
 2
 3  import java.util.HashMap;
 4  import javax.jws.WebService; // For annotation @WebService
 5  import javax.jws.WebMethod; // For annotation @WebMethod
 6
 7  @WebService(name = "ScoreService", serviceName = "ScoreWebService")
 8  public class ScoreService {
 9    // Stores scores in a map indexed by name
10    private HashMap<String, Double> scores =
11      new HashMap<String, Double>();
12
13    public ScoreService() {
14      scores.put("John", 90.5);
15      scores.put("Michael", 100.0);
16      scores.put("Michelle", 98.5);
17    }
18
19    @WebMethod(operationName = "findScore")
20    public double findScore(String name) {
21      Double d = scores.get(name);
22
23      if (d == null) {
24        System.out.println("Student " + name + " is not found ");
25        return -1;
26      }
27      else {
28        System.out.println("Student " + name + "\'s score is "
29          + d.doubleValue());
30        return d.doubleValue();
31      }
32    }
33  }
```

Lines 4–5 import the annotations used in the program in lines 7 and 19. Annotation is a new feature in Java, which enables you to simplify coding. The compiler will automatically generate the code for the annotated directives. So, it frees the programmer from writing the detailed *boilerplate code* that could be generated mechanically. The annotation (line 7)

```
@WebService(name = "ScoreService", serviceName = "ScoreWebService")
```

tells the compiler that the class `ScoreService` is associated with the Web service named `ScoreWebService`.

The annotation (line 19)

```
@WebMethod(operationName = "findScore")
```

indicates that `findScore` is a method that can be invoked from a client.

The `findScore` method returns a score if the name is in the hash map. Otherwise, it returns `-1.0`.

You can manually type the code for the service, or create it from the Design tab, as shown in Figure 41.2.

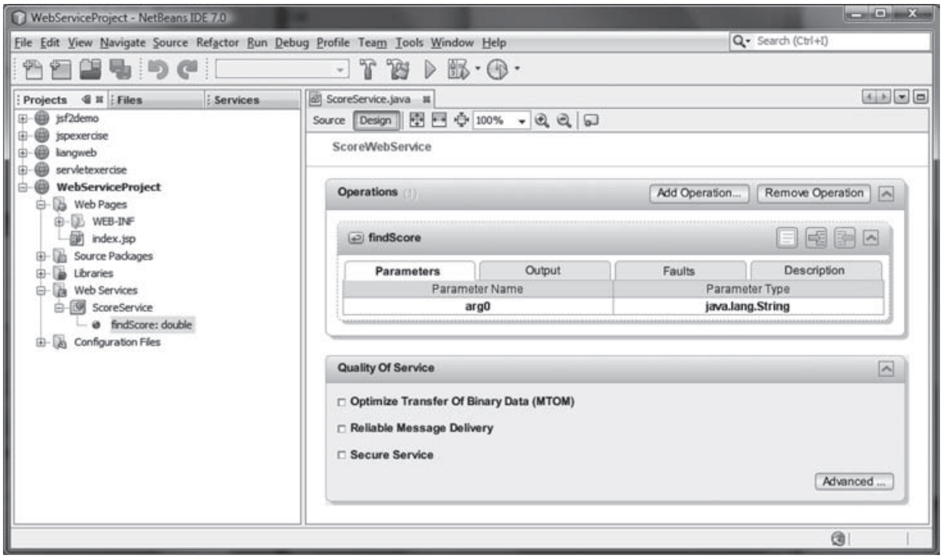

WebServiceProject - NetBeans IDE 7.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
Projects
Files
Services
jsf2demo
jspexercise
liangweb
servletexercise
WebServiceProject
Web Pages
WEB-INF
index.jsp
Source Packages
Libraries
Web Services
ScoreService
findScore: double
Configuration Files
ScoreService.java
Source
Design
100%
ScoreWebService
Operations (1)
Add Operation...
Remove Operation
findScore
Parameters
Output
Faults
Description
Parameter Name
Parameter Type
arg0
java.lang.String
Quality Of Service
☐ Optimize Transfer Of Binary Data (MTOM)
☐ Reliable Message Delivery
☐ Secure Service
Advanced ...

FIGURE 41.2 The services can also be created from the Design pane.

41.3 Deploying and Testing Web Services

Key Point

Deploying a Web service is to make it available on the Internet for other programs to use.

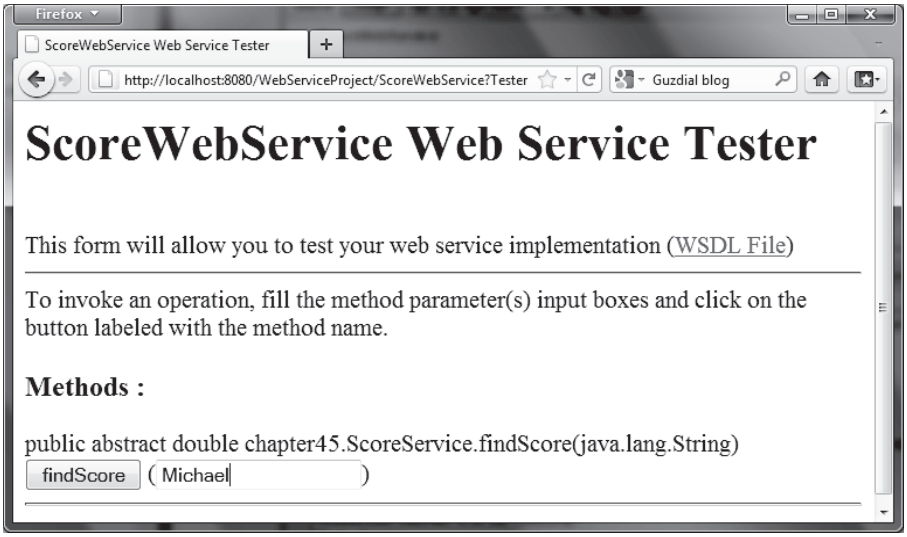
After a Web service is created, you need to deploy it for clients to use. Deploying Web services is also known as *publishing Web services*. To deploy it, right-click the `WebServiceProject` in the Project to display a context menu and choose **`Deploy`**. This command will first undeploy the service if it was deployed and then redeploy it.

Now you can test the Web service by entering the follow URL in a browser, as shown in Figure 41.3.

http://localhost:8080/WebServiceProject/ScoreWebService?Tester

Note that `ScoreWebService` is the name you specified in line 7 in Listing 41.1. This Web service has only one remote method named `findScore`. You can define an unlimited number of remote methods in a Web service class. If so, all these methods will be displayed in the test page.

To test the `findScore` method, enter `Michael` and click *findScore*. You will see that the method returns `100.0`, as shown in Figure 41.4.


Firefox
ScoreWebService Web Service Tester
http://localhost:8080/WebServiceProject/ScoreWebService?Tester
Guzdial blog

ScoreWebService Web Service Tester

This form will allow you to test your web service implementation (WSDL File)

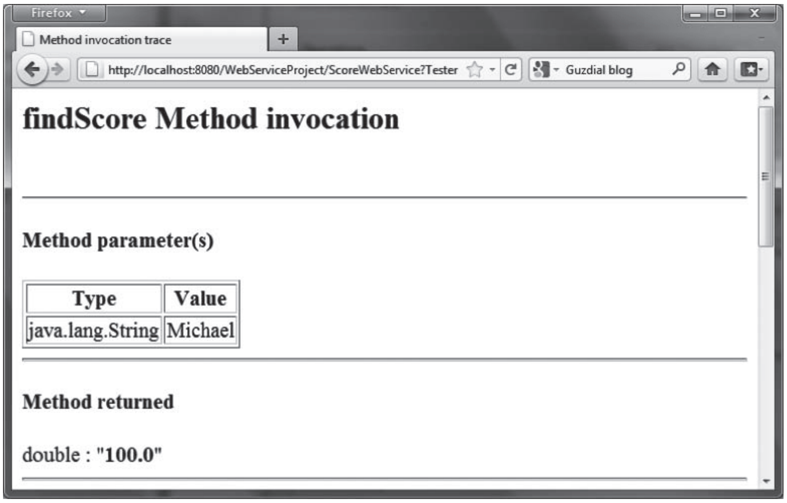
To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

Methods :

public abstract double chapter45.ScoreService.findScore(java.lang.String)

findScore (Michael)

FIGURE 41.3 The test page enables you to test Web services.


Firefox
Method invocation trace
http://localhost:8080/WebServiceProject/ScoreWebService?Tester
Guzdial blog

findScore Method invocation

Method parameter(s)

Type	Value
java.lang.String	Michael

Method returned

double : "100.0"

FIGURE 41.4 The method returns a test value.

Note

If your computer is connected to the Internet, you can test Web services from another computer by entering the following URL:

http://host:8080/WebServiceProject/ScoreWebService?Tester

Where *host* is the host name or IP address of the server on which the Web service is running. On Windows, you can find your IP address by typing the command **`ipconfig`**.

Note

If you are running the server on Windows, the firewall may prevent remote clients from accessing the service. To enable it, do the following:

1. In the Windows control panel, click Windows Firewall to display the Windows Firewall dialog box.

2. In the Advanced tab, double-click Local Area Connection to display the Advanced Settings dialog box. Check *Web Server (HTTP)* to enable HTTP access to the server.
3. Click *OK* to close the dialog box.

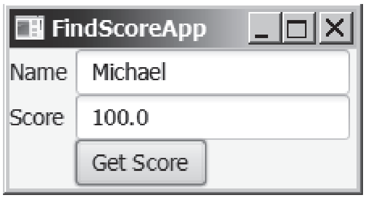
41.4 Consuming Web Services

Key Point

Consuming a Web service is for a client to use a Web service.

After a Web service is published, you can write a client program to use it. A client can be any program (standalone application, servlet/JSP/JSF application, or another Web service) and written in any language.

We will use NetBeans to create a Web service client. Our client is a GUI application. The application simply lets the user enter a name and displays the score, as shown in Figure 41.5.



Name: Michael
Score: 100.0

FIGURE 41.5 The client uses the Web service to find scores.

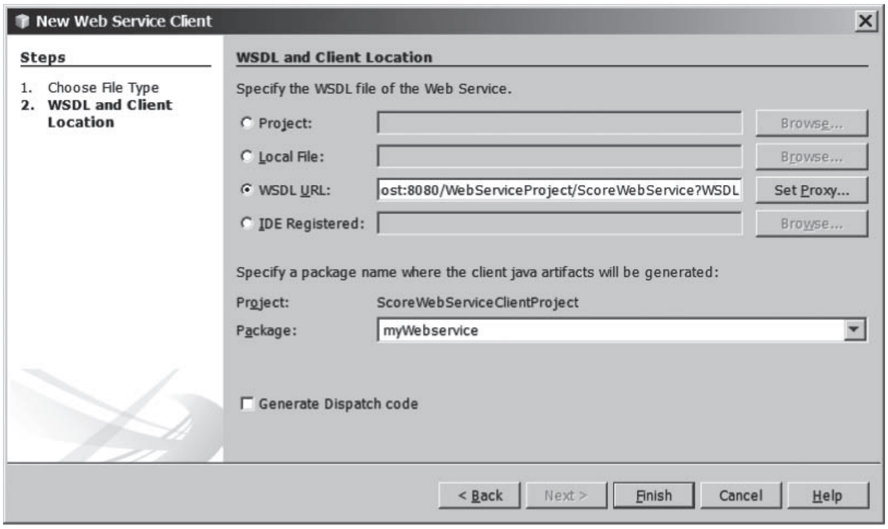
Let us create a project for the client. The project named `ScoreWebServiceClientProject` can be created as follows:

1. Choose **`File, New Project`** to display the New Project dialog box.
2. In the New Project dialog box, choose **`Java`** in the Categories pane and choose **`Java Application`** in the Projects pane. Click *Next* to display the New Java Application dialog box.
3. Enter `ScoreWebServiceClientProject` as the project name, specify the location where you want the project to be stored, and uncheck the *Create Main Class* check box. Click *Finish* to create the project.

You need to create a Web service reference to this project. The reference will enable you to create a proxy object to interact with the Web service. Here are the steps to create a Web service reference:

1. Right-click the `ScoreWebServiceClientProject` in the Project pane to display a context menu. Choose **`New, Web Service Client`** to display the New Web Service Client dialog box, as shown in Figure 41.6.
2. Check the *WSDL URL* radio button and enter http://localhost:8080/WebServiceProject/ScoreWebService?WSDL in the WSDL URL field.
3. Enter `myWebservice` in the package name field. Click *Finish* to generate the Web service reference.

Now you will see `ScoreWebService` created in the Web Service References folder in the Projects tab. The IDE has generated many supporting files for the reference. You can view all the generated .java files from the Files tab in the project pane, as shown in Figure 41.7. These files will be used by the proxy object to interact with the Web service.


New Web Service Client
Steps
1. Choose File Type
2. WSDL and Client Location
WSDL and Client Location
○ Project:
○ Local File:
● WSDL URL:
○ IDE Registered:
Browse...
Browse...
Set Proxy...
Browse...
☐ Generate Dispatch code
< Back
Next >
Finish
Cancel
Help

Specify the WSDL file of the Web Service.

WSDL URL: ost:8080/WebServiceProject/ScoreWebService?WSDL

Specify a package name where the client java artifacts will be generated:

Project: ScoreWebServiceClientProject

Package: myWebservice

FIGURE 41.6 The New Web Service Client dialog box creates a Web service reference.

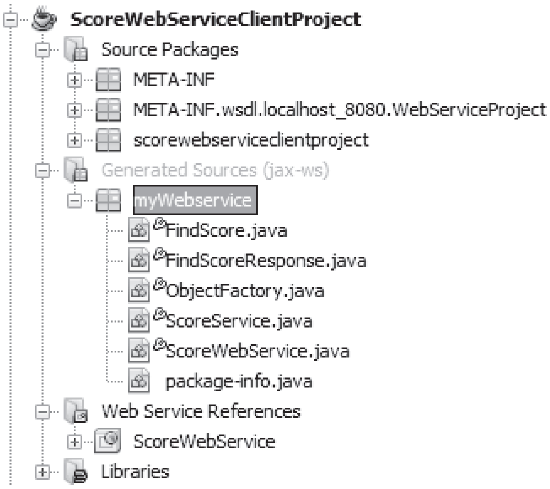

ScoreWebServiceClientProject
Source Packages
META-INF
META-INF.wsdl.localhost_8080.WebServiceProject
scorewebserviceclientproject
Generated Sources (jax-ws)
myWebservice
FindScore.java
FindScoreResponse.java
ObjectFactory.java
ScoreService.java
ScoreWebService.java
package-info.java
Web Service References
ScoreWebService
Libraries

FIGURE 41.7 You can see the automatically generated boilerplate code for Web services in the Generated Sources folder in the client's project.

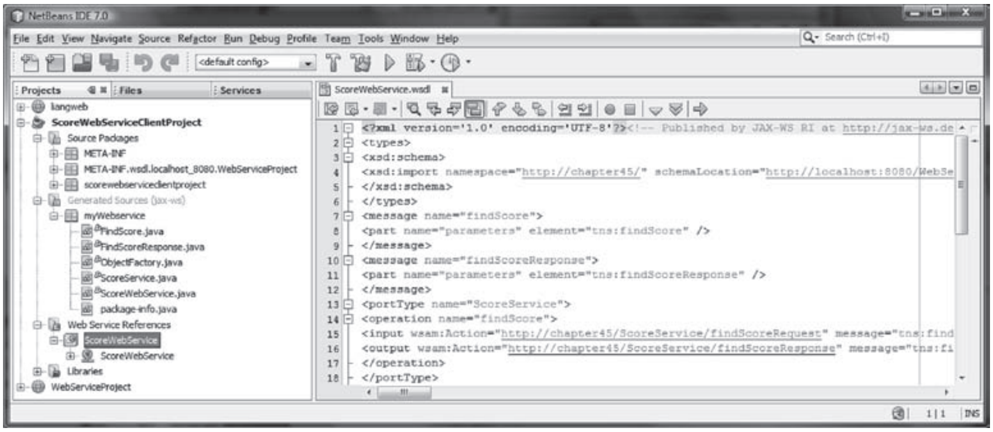
Note

When you created a Web service reference, you entered a WSDL URL, as shown in Figure 41.6. This creates a .wsdl file. In this case, it is named ScoreWebService.wsdl under the Web Service References folder, as shown in Figure 41.8. So *what is WSDL?* WSDL stands for *Web Service Description Language*. A .wsdl file is an XML file that describes the available Web service to the client—i.e., the remote methods, their parameters and return value types, and so on.

Note

If the Web service is modified, you need to refresh the reference for the client. To do so, right-click the Web service node under Web Service References to display a context menu and choose **`Refresh Client`**.

Now you are ready to create a client for the Web service. Right-click the `ScoreWebServiceClientProject` node in the Project pane to display a context menu, and choose **`New, Class`** to create a Java client named `FindScoreApp` in package `chapter41`, as shown in Listing 41.2.


NetBeans IDE 7.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Projects
Files
Services
langweb
ScoreWebServiceClientProject
Source Packages
META-INF
META-INF.wsdl.localhost_8080.WebServiceProject
scorewebserviceclientproject
Generated Sources (jax-ws)
myWebservice
FindScore.java
FindScoreResponse.java
ObjectFactory.java
ScoreService.java
ScoreWebService.java
package-info.java
Web Service References
ScoreWebService
ScoreWebService
Libraries
WebServiceProject
ScoreWebService.wsdl
1 | 1
INS

```
<?xml version='1.0' encoding='UTF-8'?><!-- Published by JAX-WS RI at http://jax-ws.de
<types>
<xsd:schema>
<xsd:import namespace="http://chapter45/" schemaLocation="http://localhost:8080/WebSe
</xsd:schema>
</types>
<message name="findScore">
<part name="parameters" element="tns:findScore" />
</message>
<message name="findScoreResponse">
<part name="parameters" element="tns:findScoreResponse" />
</message>
<portType name="ScoreService">
<operation name="findScore">
<input wsam:Action="http://chapter45/ScoreService/findScoreRequest" message="tns:find
<output wsam:Action="http://chapter45/ScoreService/findScoreResponse" message="tns:fi
</operation>
</portType>
```

FIGURE 41.8 The .wsdl file describes Web services to clients.

LISTING 41.2 FindScoreApp.java

```java
package  chapter41;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
import myWebservice.ScoreWebService;
import myWebservice.ScoreService;

public class FindScoreApp extends Application {
  // Declare a service object and a proxy object
  private ScoreWebService scoreWebService = new ScoreWebService();
  private ScoreService proxy
    = scoreWebService.getScoreServicePort();

  private Button btGetScore = new Button("Get Score");
  private TextField tfName = new TextField();
  private TextField tfScore = new TextField();

  public void  start(Stage primaryStage) {
    GridPane gridPane = new GridPane();
    gridPane.setHgap(5);
    gridPane.add(new Label("Name"), 0, 0);
    gridPane.add(new Label("Score"), 0, 1);
    gridPane.add(tfName, 1, 0);
    gridPane.add(tfScore, 1, 1);
    gridPane.add(btGetScore, 1, 2);

    // Create a scene and place the pane in the stage
    Scene scene = new Scene(gridPane, 250, 250);
    primaryStage.setTitle("FindScoreApp"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

38      btGetScore.setOnAction(e -> getScore());
39    }
40
41    private void getScore() {
42      try {
43        // Get student score
44        double score = proxy.findScore(tfName.getText().trim());
45
46        // Display the result
47        if  (score < 0)
48          tfScore.setText("Not found");
49        else
50          tfScore.setText(new Double(score).toString());
51        }
52        catch(Exception ex) {
53          ex.printStackTrace();
54      }
55    }
56  }
```

The program creates a Web service object (line 11) and creates a proxy object (line 12) to interact with the Web service.

To find a score for a student, the program invokes the remote method `findScore` on the proxy object (line 39).

41.5 Passing and Returning Arguments

The Simple Object Access Protocol (SOAP) can be used to send and return values to and from a Web service.

Key Point

In the preceding example, a Web service client that you created invokes the `findScore` method with a string argument, and the Web service executes the method and returns a score as a `double` value. How does this work? It is the *Simple Object Access Protocol* (SOAP) that facilitates communications between the client and the server.

SOAP is based on XML. The message between the client and the server is described in XML. Figure 41.9 shows the SOAP request and SOAP response for the `findScore` method.

When invoking the `findScore` method, a *SOAP request* is sent to the server. The request contains the information about the method and the argument. As shown in Figure 41.9, the XML text

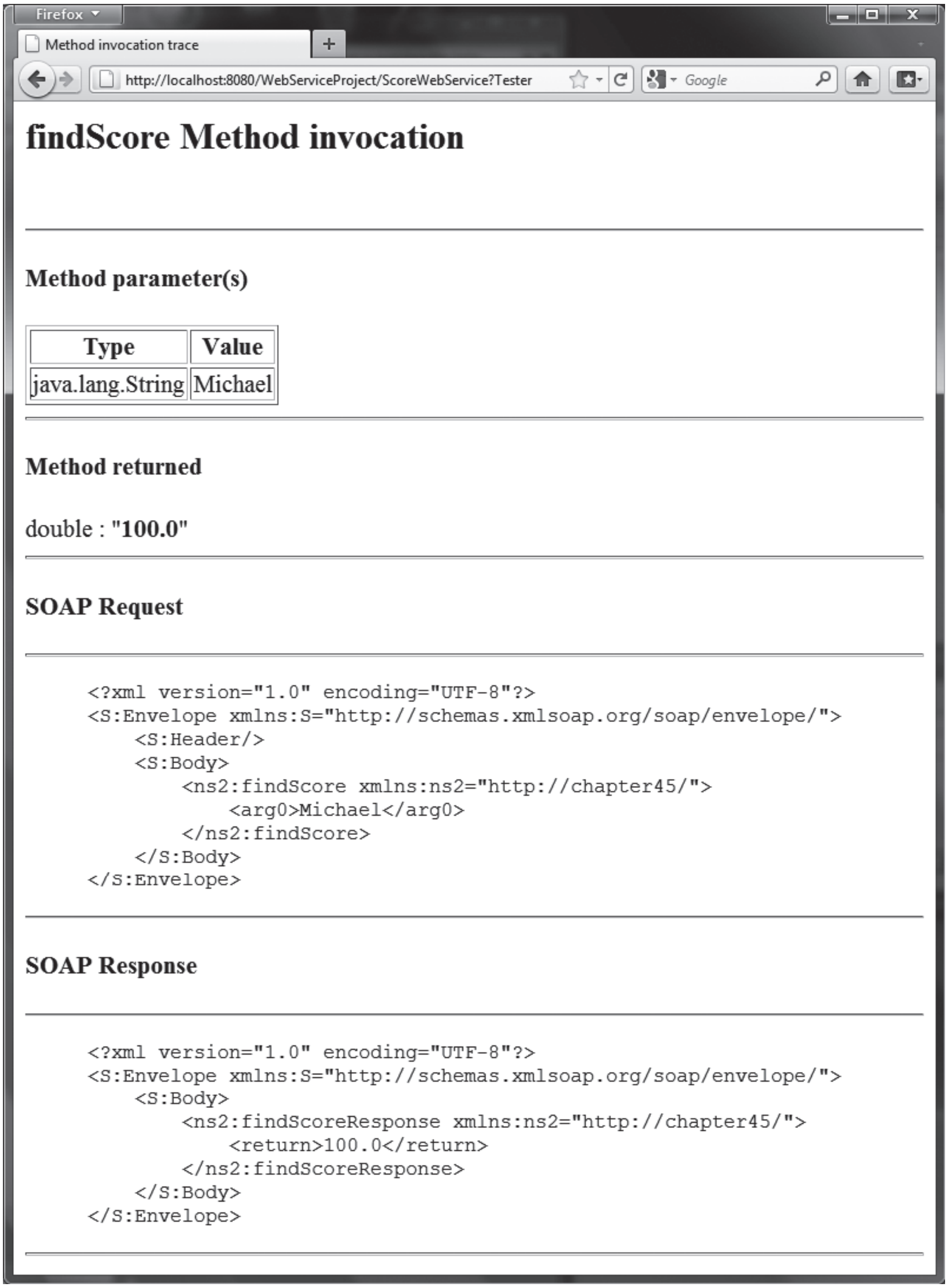
```
<ns1:findScore>
   <arg0>Michael</arg0>
</ns1:findScore>
```

specifies that the method `findScore` is called with argument `Michael`.

Upon receiving the SOAP request, the Web service parses it. After parsing it, the Web service invokes an appropriate method with specified arguments (if any) and sends the response back in a *SOAP response*. As shown in Figure 41.9, the XML text

```
<ns1:findScoreResponse>
   <return>100.0</return>
</ns1:findScoreResponse>
```

specifies that the method returns `100.0`.


Firefox
Method invocation trace
http://localhost:8080/WebServiceProject/ScoreWebService?Tester
Google

findScore Method invocation

Method parameter(s)

Type	Value
java.lang.String	Michael

Method returned

double : **"100.0"**

SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header/>
    <S:Body>
        <ns2:findScore xmlns:ns2="http://chapter45/">
            <arg0>Michael</arg0>
        </ns2:findScore>
    </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:findScoreResponse xmlns:ns2="http://chapter45/">
            <return>100.0</return>
        </ns2:findScoreResponse>
    </S:Body>
</S:Envelope>
```

FIGURE 41.9 The client request and server response are described in XML.

The proxy object receives the SOAP response from the Web service and parses it. This process is illustrated in Figure 41.10.

Can you pass an argument of any type between a client and a Web service? No. SOAP supports only primitive types, wrapper types, arrays, `String`, `Date`, `Time`, `List`, and several other types. It also supports certain custom classes. An object that is sent to or from a server is serialized into XML. The process of serializing/deserializing objects, called

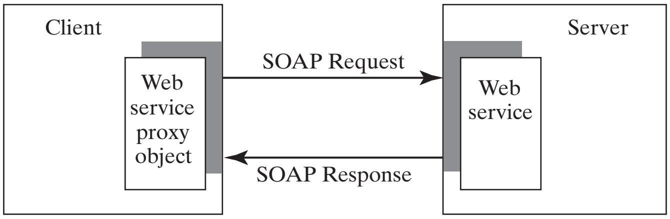


FIGURE 41.10 A proxy object sends SOAP requests and receives SOAP responses.

XML serialization/deserialization, is performed automatically. For a custom class to be used with Web methods, the class must meet the following requirements:

1. The class must have a no-arg constructor.
2. Instance variables that should be serialized must have public get and set methods. The classes of these variables must be supported by SOAP.

To demonstrate how to pass an object argument of a custom class, Listing 41.3 defines a Web service class named `AddressService` with two remote methods:

- `getAddress(String firstName, String lastName)` that returns an `Address` object for the specified `firstName` and `lastName`.
- `storeAddress(Address address)` that stores a `Student` object to the database.

Address information is stored in a table named `Address` in the database. The `Address` class was defined in Listing 42.12, Address.java. An `Address` object can be passed to or returned from a remote method, since the `Address` class has a no-arg constructor with get and set methods for all its properties.

Here are the steps to create a Web service named `AddressService` and the `Address` class in the project.

1. Right-click the `WebServiceProject` node in the project pane to display a context menu. Choose **`New, Web Service`** to display the New Web Service dialog box.
2. In the Web Service Name field, enter `AddressService`. In the Package field, enter `chapter41`. Click *Finish* to create the service class.
3. Right-click the `WebServiceProject` node in the project pane to display a context menu. Choose **`New, Java Class`** to display the New Java Class dialog box.
4. In the Class Name field, enter `Address`. In the Package field, enter `chapter37`. Click *Finish* to create the class.

The `Address` class is the same as shown in Listing 37.12. Complete the `AddressService` class as shown in Listing 41.3.

LISTING 41.3 AddressService.java

```java
package chapter41;

import chapter37.Address;
import java.sql.*;
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService(name = "AddressService",
  serviceName = "AddressWebService")
```

```
public class AddressService {
  // statement1 for retrieving an address and statement2 for storing
  private PreparedStatement statement1;

  // statement2 for storing an address
  private PreparedStatement statement2;

  public AddressService() {
    initializeJdbc();
  }

  @WebMethod(operationName = "getAddress")
  public Address getAddress(String firstName, String lastName) {
    try {
      statement1.setString(1, firstName);
      statement1.setString(2, lastName);
      ResultSet resultSet = statement1.executeQuery();

      if (resultSet.next()) {
        Address address = new Address();
        address.setFirstName(resultSet.getString("firstName"));
        address.setLastName(resultSet.getString("lastName"));
        address.setMi(resultSet.getString("mi"));
        address.setTelephone(resultSet.getString("telephone"));
        address.setFirstName(resultSet.getString("email"));
        address.setCity(resultSet.getString("telephone"));
        address.setState(resultSet.getString("state"));
        address.setZip(resultSet.getString("zip"));
        return address;
      }
      else
        return null;
    } catch (SQLException ex) {
      ex.printStackTrace();
    }

    return null;
  }

  @WebMethod(operationName = "storeAddress")
  public void storeAddress(Address address) {
    try {
      statement2.setString(1, address.getLastName());
      statement2.setString(2, address.getFirstName());
      statement2.setString(3, address.getMi());
      statement2.setString(4, address.getTelephone());
      statement2.setString(5, address.getEmail());
      statement2.setString(6, address.getStreet());
      statement2.setString(7, address.getCity());
      statement2.setString(8, address.getState());
      statement2.setString(9, address.getZip());
      statement2.executeUpdate();
    } catch (SQLException ex) {
    ex.printStackTrace();
    }
  }

  /** Initialize database connection */
  public void initializeJdbc() {
```

```
69      try {
70        Class.forName("com.mysql.jdbc.Driver");
71
72        // Connect to the sample database
73        Connection connection = DriverManager.getConnection(
74          "jdbc:mysql://localhost/javabook", "scott", "tiger");
75
76        statement1 = connection.prepareStatement(
77          "select * from Address where firstName = ? and lastName = ?");
78        statement2 = connection.prepareStatement(
79          "insert into Address "  +
80          "(lastName, firstName, mi, telephone, email, street, city, "
81          + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
82      } catch (Exception ex) {
83        ex.printStackTrace();
84      }
85    }
86  }
```

The new Web service is named `AddressWebService` (line 9) for the `AddressService` class.

When the service is deployed, the constructor (lines 17–19) of `AddressWebService` is invoked to initialize a database connection and create prepared `statement1` and `statement2` (lines 68–85).

The `findAddress` method searches the address in the `Address` table for the specified `firstName` and `lastName`. If found, the address information is returned in an `Address` object (lines 29–38). Otherwise, the method returns `null` (line 41).

The `storeAddress` method stores the address information from the `Address` object into the database (lines 52–61).

Note
Don't forget that you have to add the MySQL library to the `WebServiceProject` for this example to run.

Before you can use the service, deploy it. Right-click the `WebServiceProject` node in the Project to display a context menu and choose **`Deploy`**.

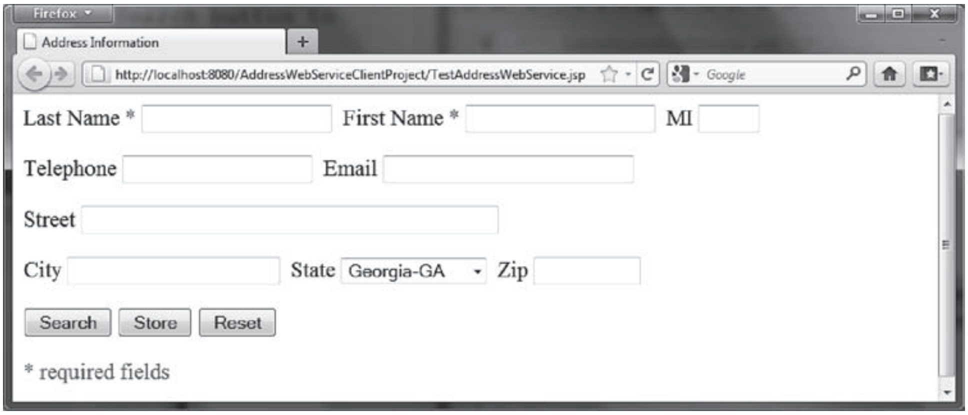
Now you are ready to develop a Web client that uses the `AddressWebService`. The client is a JSP program, as shown in Figure 41.11. The program has two functions. First, the user can enter the last name and first name and click the *Search* button to search for a record, as shown in Figure 41.12. Second, the user can enter the complete address information and click the *Store* button to store the information to the database, as shown in Figure 41.13.

Let us create a project for the client. The project named `AddressWebServiceClientProject` can be created as follows:

1. Choose **`File, New Project`** to display the New Web Application dialog box. In the New Web Application dialog box, choose **`Java Web`** in the Categories pane and choose **`Web Application`** in the Projects pane. Click *Next* to display the Name and Location dialog box.

2. Enter `AddressWebServiceClientProject` as the project name, specify the location where you want the project to be stored, and uncheck the *Set as Main Project* check box. Click *Next* to display the Server and Settings dialog box.

3. Choose **`GlassFish Server 4`** in the Server field, and **`Java EE 7 Web`** as in the Java EE Version field, and click Finish to create the project.


Firefox
Address Information
http://localhost:8080/AddressWebServiceClientProject/TestAddressWebService.jsp
Google
Search
Store
Reset

Last Name * First Name * MI

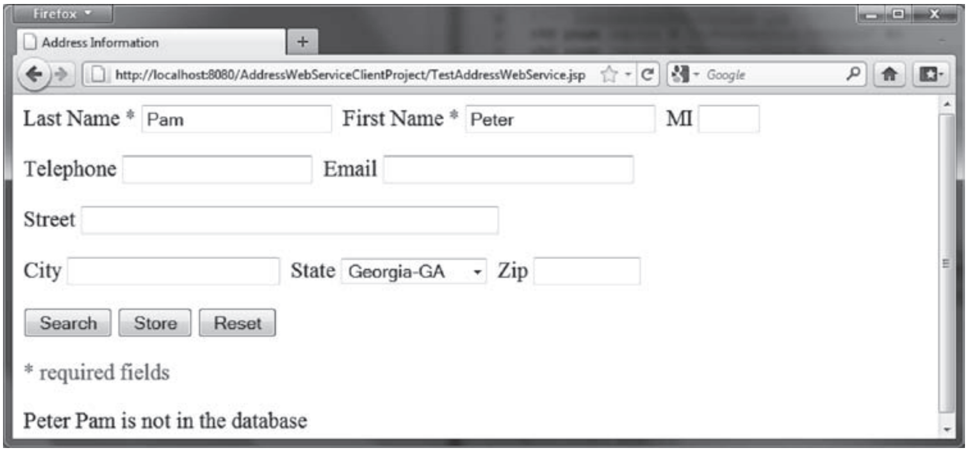
Telephone Email

Street

City State Georgia-GA Zip

* required fields

FIGURE 41.11 The `TestAddressWebService` page allows the user to search and store addresses.


Firefox
Address Information
http://localhost:8080/AddressWebServiceClientProject/TestAddressWebService.jsp
Google
Search
Store
Reset

Last Name * Pam First Name * Peter MI

Telephone Email

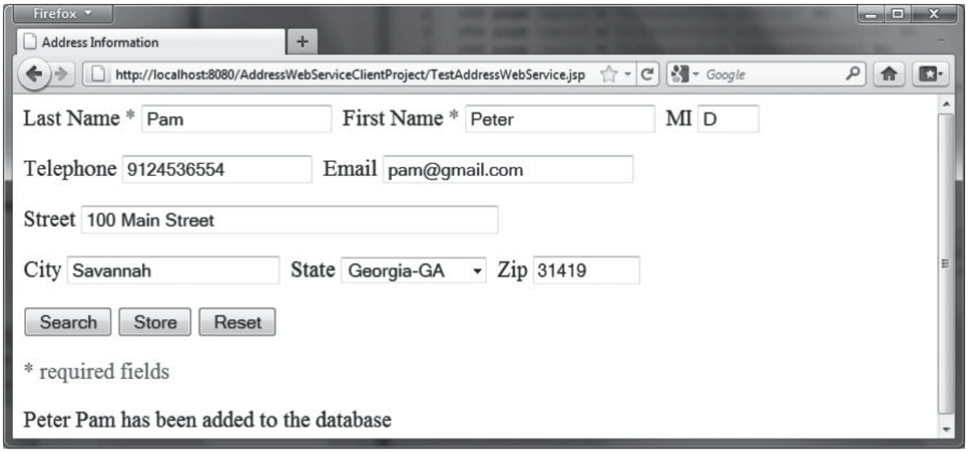
Street

City State Georgia-GA Zip

* required fields

Peter Pam is not in the database

FIGURE 41.12 The Search button finds and displays an address.


Firefox
Address Information
http://localhost:8080/AddressWebServiceClientProject/TestAddressWebService.jsp
Google
Search
Store
Reset

Last Name * Pam First Name * Peter MI D

Telephone 9124536554 Email pam@gmail.com

Street 100 Main Street

City Savannah State Georgia-GA Zip 31419

* required fields

Peter Pam has been added to the database

FIGURE 41.13 The Store button stores the address to the database.

You need to create a Web service reference to this project. The reference will enable you to create a proxy object to interact with the Web service. Here are the steps to create a Web service reference:

1. Right-click the `AddressWebServiceClientProject` node in the Project pane to display a context menu. Choose **`New, Web Service Client`** to display the New Web Service Client dialog box.
2. Check the *WSDL URL radio* button and enter http://localhost:8080/WebServiceProject/AddressWebService?WSDL in the WSDL URL field.
3. Enter `myWebservice` in the package name field and choose **`JAX-WS`** as the JAX version. Click *Finish* to generate the Web service reference.

Now a reference to `AddressWebService` is created. Note that this process also copies `Address.java` to the client project, as shown in Figure 41.14.

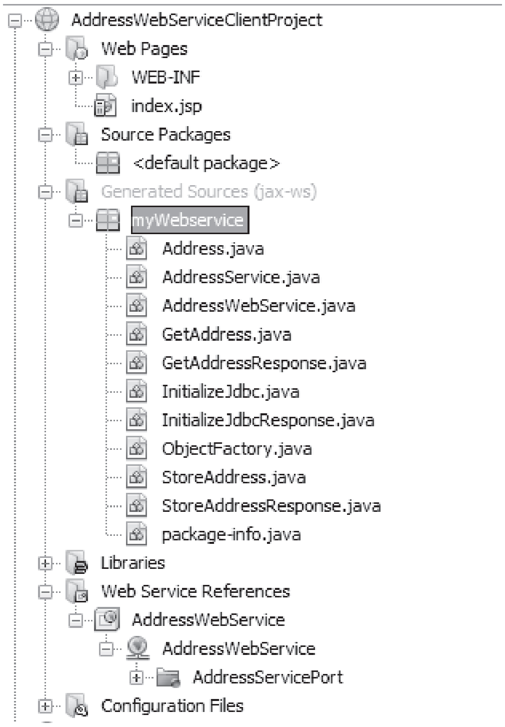

AddressWebServiceClientProject
Web Pages
WEB-INF
index.jsp
Source Packages
<default package>
Generated Sources (jax-ws)
myWebservice
Address.java
AddressService.java
AddressWebService.java
GetAddress.java
GetAddressResponse.java
InitializeJdbc.java
InitializeJdbcResponse.java
ObjectFactory.java
StoreAddress.java
StoreAddressResponse.java
package-info.java
Libraries
Web Service References
AddressWebService
AddressWebService
AddressServicePort
Configuration Files

FIGURE 41.14 The Address.java is automatically copied to the Web service client reference package.

Create a JSP named `TestAddressWebService` in the `AddressWebServiceClientProject` project, as shown in Listing 41.4.

LISTING 41.4 `TestAddressWebService.jsp`

```
<!-- TestAddressWebService.jsp -->
<%@ page import = "myWebservice.Address" %>
<%@ page import = "myWebservice.AddressWebService" %>
<%@ page import = "myWebservice.AddressService" %>
<jsp:useBean id = "addressId"
  class = "myWebservice.Address" scope = "session"></jsp:useBean>
<jsp:setProperty name = "addressId" property = "*" />

```

```
<html>
<head>
  <title>Address Information</title>
</head>
<body>
  <form method = "post" action = "TestAddressWebService.jsp">
  Last Name <font color = "#FF0000">*</font>
  <input type = "text" name = "lastName"
    <%if (addressId.getLastName() != null) {
      out.print("value = \"" + addressId.getLastName() + "\"");}%>
    size = "20" />&nbsp;

  First Name <font color = "#FF0000">*</font>
  <input type = "text" name = "firstName"
    <%if (addressId.getFirstName() != null) {
      out.print("value = \"" + addressId.getFirstName() + "\"");}%>
    size = "20" />&nbsp;

  MI
  <input type = "text" name = "mi"
    <%if (addressId.getMi() != null) {
      out.print("value = \"" + addressId.getMi() + "\" "); } %>
    size = "3" />&nbsp;

  <p>Telephone
  <input type = "text" name = "telephone"
    <%if (addressId.getTelephone() != null) {
      out.print("value = \"" + addressId.getTelephone() + "\" ");}%>
    size = "20" />&nbsp;

   Email
   <input type = "text" name = "email"
     <%if (addressId.getEmail() != null) {
       out.print("value = \"" + addressId.getEmail() + "\" ");}%>
     size = "28" />&nbsp;
   </p>

   <p>Street
   <input type = "text" name = "street"
     <%if (addressId.getStreet() != null) {
       out.print("value = \"" + addressId.getStreet() + "\" ");}%>
     size = "50" />&nbsp;
   </p>

   <p>City
   <input type = "text" name = "city"
     <%if (addressId.getCity() != null) {
       out.print("value = \"" + addressId.getCity() + "\" ");}%>
   size = "23" />&nbsp;

  State
  <select size = "1" name = "state">
    <option value = "GA">Georgia-GA</option>
    <option value = "OK">Oklahoma-OK</option>
    <option value = "IN">Indiana-IN</option>
  </select>&nbsp;

  Zip
  <input type = "text" name = "zip"
    <%if (addressId.getZip() != null) {
      out.print("value = \"" + addressId.getZip() + "\" "); } %>
```

```
 70       size = "9" />&nbsp;
 71     </p>
 72 
 73     <p><input type = "submit" name = "Submit" value = "Search">
 74       <input type = "submit" name = "Submit" value = "Store">
 75       <input type = "reset" value = "Reset">
 76     </p>
 77     </form>
 78     <p><font color = "#FF0000">* required fields</font></p>
 79 
 80     <%
 81     if (request.getParameter("Submit") != null) {
 82       AddressWebService addressWebService = new AddressWebService();
 83       AddressService proxy = addressWebService.getAddressServicePort();
 84 
 85       if (request.getParameter("Submit").equals("Store")) {
 86         proxy.storeAddress(addressId);
 87         out.println(addressId.getFirstName() + " " +
 88           addressId.getLastName() + " has been added to the database");
 89       }
 90       else if (request.getParameter("Submit").equals("Search")) {
 91         Address address = proxy.getAddress(addressId.getFirstName(),
 92           addressId.getLastName());
 93         if (address == null)
 94           out.print(addressId.getFirstName() + " " +
 95             addressId.getLastName() + " is not in the database");
 96         else
 97           addressId = address;
 98       }
 99     }
100     %>
101   </body>
102   </html>
```

Lines 2–4 import the classes for the JSP page. The `Address` class (line 2) was created in the `WebServiceProject` and was automatically copied to the `AddressWebServiceClientProjec` project when a Web service reference for `AddressWebService` was created. A JavaBeans object for `Address` was created and associated with input parameters in lines 5–7.

The UI interface was laid in the form (lines 14–77). The action for the two buttons *Search* and *Store* invokes the same page TestAddressWebService.jsp (line 14).

When a button is clicked, a proxy object for `AddressWebService` is obtained (lines 82–83). For the *Store* button, the proxy object invokes the `storeAddress` method to add an address to the database (line 86). For the *Search* button, the proxy object invokes the `getAddress` method to return an address (lines 91–92). If no address is found for the specified first and last names, the returned address is `null` (line 93).

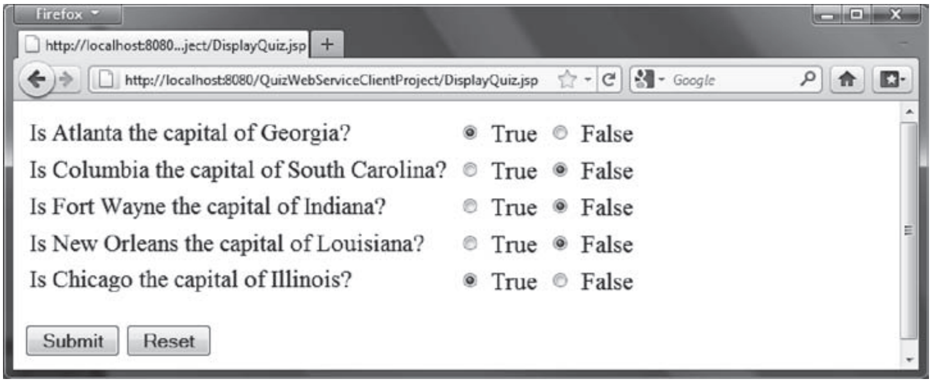
41.6 Web Service Session Tracking

You can use the `HttpSession` *interface to session tracking for Web.*

Key Point

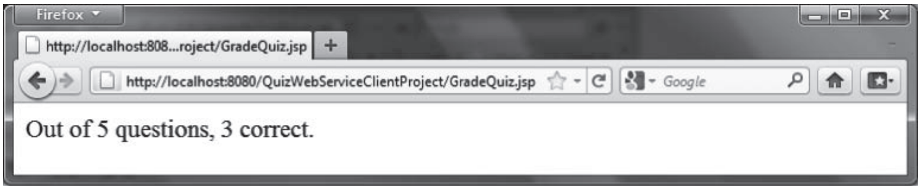
Section 37.8.3, "Session Tracking Using the Servlet API," introduced session tracking for servlets using the `javax.servlet.http.HttpSession` interface. You can use `HttpSession` to implement session tracking for Web services. To demonstrate this, consider an example that generates random True/False questions for the client and grades the answers on these questions for the client.

The Web client consists of two JSP pages: DisplayQuiz.jsp and GradeQuiz.jsp. The `DisplayQuiz` page invokes the service method `getQuestion()` to display the questions, as shown in Figure 41.15. When you click the *Submit* button, the program invokes the service


Firefox
http://localhost:8080...ject/DisplayQuiz.jsp
http://localhost:8080/QuizWebServiceClientProject/DisplayQuiz.jsp
Google
Submit
Reset

Is Atlanta the capital of Georgia? ● True ○ False
Is Columbia the capital of South Carolina? ○ True ● False
Is Fort Wayne the capital of Indiana? ○ True ● False
Is New Orleans the capital of Louisiana? ○ True ● False
Is Chicago the capital of Illinois? ● True ○ False

FIGURE 41.15 The Submit button submits the answers for grading.


Firefox
http://localhost:808...roject/GradeQuiz.jsp
http://localhost:8080/QuizWebServiceClientProject/GradeQuiz.jsp
Google

Out of 5 questions, 3 correct.

FIGURE 41.16 The answers are graded and displayed.

method `gradeQuiz` to grade the answers. The result is displayed in the `GradeQuiz` page, as shown in Figure 41.16.

Why is session tracking needed for this project? Each time a client displays a quiz, it creates a randomly reorder the quiz for the client. Each client gets a different quiz every time the `DisplayQuiz` page is refreshed. When the client submits the answer, the Web service checks the answer against the previously generated quiz. So the quiz has to be stored in the session.

For convenience, let us create the Web service class named `QuizService` in the `WebServiceProject` in package `chapter41`. Listing 41.5 gives the program.

LISTING 41.5 QuizService.java

```
package chapter41;

import javax.jws.WebMethod;
import javax.jws.WebService;
import java.util.List;
import java.util.ArrayList;
import com.sun.xml.ws.developer.servlet.HttpSessionScope;

@HttpSessionScope
@WebService(name = "QuizService", serviceName = "QuizWebService")
public class QuizService {
  private ArrayList<Object[]> quiz = new ArrayList<Object[]>();

  public QuizService() {
    // Initialize questions and answers
    quiz.add(new Object[]{
      "Is Atlanta the capital of Georgia?", true});
    quiz.add(new Object[]{
      "Is Columbia the capital of South Carolina?", true});
    quiz.add(new Object[]{
      "Is Fort Wayne the capital of Indiana?", false});
22      quiz.add(new Object[]{
23        "Is New Orleans the capital of Louisiana?", false});
24      quiz.add(new Object[]{
25        "Is Chicago the capital of Illinois?", false});
26
27      // Shuffle to generate a random quiz for a client
28      java.util.Collections.shuffle(quiz);
29    }
30
31    @WebMethod(operationName = "getQuestions")
32    public java.util.List<String> getQuestions() {
33
34      // Extract questions from quiz
35      List<String> questions = new ArrayList<String>();
36      for  (int i = 0; i < quiz.size(); i++) {
37        questions.add((String)(quiz.get(i)[0]));
38      }
39
40      return questions; // Return questions in the quiz
41    }
42
43    @WebMethod(operationName = "gradeQuiz")
44    public List<Boolean> gradeQuiz(List<Boolean> answers) {
45      List<Boolean> result = new ArrayList<Boolean>();
46      for (int i = 0; i < quiz.size(); i++)
47        result.add(quiz.get(i)[1] == answers.get(i));
48
49      return result;
50    }
51  }
```

The Web service class named `QuizService` contains two methods `getQuestions` and `gradeQuiz`. The new Web service is named `QuizWebService` (line 10).

The annotation `@HttpSessionScope` (line 9) is new in JAX-WS 2.2, which enables the Web service automatically maintains a separate instance for each client session. To use this annotation, you have add JAX-WS 2.2 into your project's library. This can be done by clicking the **`Library`** node in the project and select **`Add Library`**.

Assume that five True/False questions are available from the service. The quiz is stored in an `ArrayList` (lines 16–25).

Each element in the list is an array with two values. The first value is a string that describes the question and the second is a Boolean value indicating whether the answer should be true or false.

A new quiz is generated in the constructor and the quiz is shuffled using the shuffle method in the Collections class (line 28).

The `getQuestions` method (lines 31–40) returns questions in a list. The questions are extracted from the quiz (lines 34–37) and are returned (line 39).

The `gradeQuiz` method (lines 42–49) checks the `answers` from the client with the answers in the quiz. The client's answers are compared with the key, and the result of the grading is stored in a list. Each element in the list is a Boolean value that indicates whether the answer is correct or incorrect (lines 44–46).

After creating and publishing the Web service, let us create a project for the client. The project named `QuizWebServiceClientProject` can be created as follows:

1. Choose **`File, New Project`** to display the New Web Application dialog box.
2. In the New Web Application dialog box, choose **`Java Web`** in the Categories pane and choose **`Web Application`** in the Projects pane. Click *Next* to display the Name and Location dialog box.

3. Enter `QuizWebServiceClientProject` as the project name, specify the location where you want the project to be stored, and uncheck the *Set as Main Project* check box. Click *Next* to display the Server and Settings dialog box.
4. Choose **`GlassFish Server 4`** in the Server field, and **`Java EE 7 Web`** as in the Java EE Version field, and click Finish to create the project.

To use `QuizWebService`, you need to create a Web service client as follows:

1. Right-click the `QuizWebServiceClientProject` project in the Project pane to display a context menu. Choose **`New, Web Service Client`** to display the New Web Service Client dialog box.
2. Check the *WSDL URL radio* button and enter http://localhost:8080/WebServiceProject/QuizWebService?WSDL in the WSDL URL field.
3. Enter `myWebservice` in the Package field. Click *Finish* to create the reference for `QuizWebService`.

Now a reference to `QuizWebService` is created. You can create a proxy object to access the remote methods in `QuizService`. Listings 41.6 and 41.7 show DisplayQuiz.jsp and GradeQuiz.jsp.

LISTING 41.6 DisplayQuiz.jsp

```
<!-- DisplayQuiz.jsp -->
<%@ page import = "myWebservice.QuizWebService" %>
<%@ page import = "myWebservice.QuizService" %>
<jsp:useBean id = "quizWebService" scope = "session"
  class = "myWebservice.QuizWebService">
</jsp:useBean>

<html>
<body>
  <%
  QuizService proxy = quizWebService.getQuizServicePort();
  java.util.List<String> questions =
    (java.util.ArrayList<String>)(proxy.getQuestions());
  %>
  <form method = "post" action = "GradeQuiz.jsp">
  <table>
    <% for (int i = 0; i < questions.size(); i++) {%>
    <tr>
    <td>
      <label><%= questions.get(i) %></label>
    </td>
    <td>
      <input type = "radio" name = <%= "question" + i%>
        value = "True" /> True
    </td>
    <td>
      <input type = "radio" name = <%= "question" + i%>
      value = "False" /> False
    </td>
    </tr>
    <%}%>
  </table>
  <p><input type = "submit" name = "Submit" value = "Submit">
    <input type = "reset" value = "Reset">
```

```
35      </p>
36      </form>
37    </body>
38    </html>
```

This page generates a quiz by invoking the `getQuestions()` in lines 12–13. The questions are displayed in a table with radio buttons (lines 16–32). Clicking *Submit* invokes GradeQuiz.jsp.

LISTING 41.7 GradeQuiz.jsp

```
 1    <!-- GradeQuiz.jsp -->
 2    <%@ page import = "myWebservice.QuizWebService" %>
 3    <%@ page import = "myWebservice.QuizService" %>
 4    <jsp:useBean id = "quizWebService" scope = "session"
 5      class = "myWebservice.QuizWebService">
 6    </jsp:useBean>
 7
 8    <html>
 9    <body>
10    <%
11    QuizService proxy = quizWebService.getQuizServicePort();
12    java.util.List<String> quiz = proxy.getQuestions();
13
14    // Get the answer from the DisplayQuiz page
15    java.util.List<Boolean> answers = new java.util.ArrayList<Boolean>();
16    for (int i = 0; i < quiz.size(); i++) {
17      String trueOrFalse = request.getParameter("question" + i);
18      if (trueOrFalse.equals("True"))
19        answers.add(true); // Answered true
20      else if (trueOrFalse.equals("False"))
21        answers.add(false); // Answered false
22    }
23
24    // Grade answers
25    java.util.List<Boolean> result = proxy.gradeQuiz(answers);
26
27    // Find the correct count
28    int correctCount = 0;
29    for (int i = 0; i < result.size(); i++) {
30      if (result.get(i))
31        correctCount++;
32    }
33    %>
34
35    Out of <%= result.size() %> questions, <%= correctCount %> correct.
36    </body>
37    </html>
```

This page collects the answers passed from the HTML form from the `DisplayQuiz` page (lines 15–21), invokes the `gradeQuiz` method to grade the quiz (line 25), finds the correct count (lines 28–31), and displays the result (line 35).

Note

You need to answer all five questions before clicking the *Submit* button. A runtime error will occur if a radio button is not checked. You can fix this problem in Exercise 41.5.

Check Point

41.6.1 What is the annotation to specify a Web service? What is the annotation to specify a Web method?

41.6.2 How do you deploy a Web service in NetBeans?

41.6.3 Can you test a Web service from a client?

41.6.4 How do you create a Web service reference for a client?

41.6.5 What is WSDL? What is SOAP? What is a SOAP request? What is a SOAP response?

41.6.6 Can you pass primitive type arguments to a remote method? Can you pass any object type to a remote method? Can you pass an argument of a custom type to a remote method?

41.6.7 How do you obtain an `HttpSession` object for tracking a Web session?

41.6.8 Can you create two Web service references in one package in the same project in NetBeans?

41.6.9 What happens if you don't clone the quiz in lines 40–41 in Listing 41.5, QuizService.java?

KEY TERMS

`@WebService`	41-3
`@WebMethod`	41-3
consuming a Web service	41-2
proxy object	41-2
publishing a Web service	41-2
Web service	41-2
Web service client reference	41-15
WSDL	41-6

CHAPTER SUMMARY

1. Web services enable a Java program on one system to invoke a method in an object on another system.

2. Web services are platform and language independent. You can develop and use Web services using any language.

3. Web services run on the Web using HTTP. SOAP is a popular protocol for implementing Web services.

4. The server needs to make the service available to the client, known as *publishing a Web service*. Using a Web service from a client is known as *consuming a Web service*.

5. A client interacts with a Web service through a *proxy object*. The proxy object facilitates the communication between the client and the Web service.

6. You need to use Java annotation @WebService to annotate a Web service and use annotation @WebMethod to annotate a remote method.

7. A Web service class may have an unlimited number of remote methods.

8. After a Web service is published, you can write a client program to use it. You have to first create a Web client reference. From the reference, you create a proxy object for facilitating communication between a server and a client.

9. WSDL stands for *Web Service Description Language*. A .wsdl file is an XML file that describes the available Web service to the client—i.e., the remote methods, their parameters and return value types, and so on.

10. The message between the client and the server is described in XML. A SOAP request describes the information that is sent to the Web service and a SOAP response describes the information that is received from the Web service.

11. The objects passed between client and Web service are serialized in XML. Not all object types are supported by SOAP.

12. You can track sessions in Web services using the `HttpSession` in the same way as in servlets.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

***41.1** (*Get a score from a database table)* Suppose that the scores are stored in the `Scores` table. The table was created as follows:

```
create table Scores (name varchar(20),
  score number, permission boolean);

insert into Scores values (‘John’, 90.5, 1);
insert into Scores values ('Michael', 100, 1);
insert into Scores values ('Michelle', 100, 0);
```

Revise the `findScore` method in Listing 41.1, ScoreService.java, to obtain a score for the specified name. Note that your program does not need the `permission` column; ignore it. The next exercise will need the `permission` column.

***41.2** (*Permission to find scores)* Revise the preceding exercise so that the `findScore` method returns `-1` if permission is `false`. Add an another method named `getPermission(String name)` that returns `1`, `0`, or `-1`. The method returns `1` if the student is in the `Scores` table and permission is `true`, `0` if the student is in the `Scores` table and permission is `false`, and `-1` if the student is not in the `Scores` table.

***41.3** (*Compute loan*) You can compute a loan payment for a loan with the specified amount, the number of years, and the annual interest rate. Write a Web service with two remote methods for computing monthly payment and total payment. Write a client program that prompts the user to enter the loan amount, the number of years, and the annual interest rate.

***41.4** (*Web service visit count*) Write a Web service with a method named `getCount()` that returns the number of the times this method has been invoked from a client. Use a session to store the `count` variable.

***41.5** (*Quiz*) The user needs to answer all five questions before clicking the *Submit* button in the Quiz application in Section 41.6, Web Service Session Tracking. A runtime error will occur if a radio button is not checked. Fix this problem.

CHAPTER

42

2–4 TREES AND B-TREES

Objectives

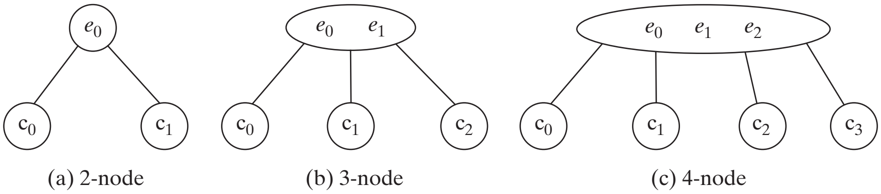
- To know what a 2–4 tree is (§42.1).
- To design the `Tree24` class that implements the `Tree` interface (§42.2).
- To search an element in a 2–4 tree (§42.3).
- To insert an element in a 2–4 tree and know how to split a node (§42.4).
- To delete an element from a 2–4 tree and know how to perform transfer and fusion operations (§42.5).
- To traverse elements in a 2–4 tree (§42.6).
- To implement and test the `Tree24` class (§§42.7–42.8).
- To analyze the complexity of the 2–4 tree (§42.9).
- To use B-trees for indexing large amount of data (§42.10).

42.1 Introduction

Key Point

A 2–4 tree, also known as a 2–3–4 tree, is a completely balanced search tree with all leaf nodes appearing on the same level.

In a 2–4 tree, a node may have one, two, or three elements. An interior *2-node* contains one element and two children. An interior *3-node* contains two elements and three children. An interior *4-node* contains three elements and four children, as shown in Figure 42.1.



(a) 2-node (b) 3-node (c) 4-node

FIGURE 42.1 An interior node of a 2–4 tree has two, three, or four children.

Each child is a sub 2–4 tree, possibly empty. The root node has no parent, and leaf nodes have no children. The elements in the tree are distinct. The elements in a node are ordered such that

$$E(c_0) < e_0 < E(c_1) < e_1 < E(c_2) < e_2 < E(c_3)$$

where $E(c_k)$ denote the elements in c_k. Figure 42.2 shows an example of a 2–4 tree. c_k is called the *left subtree* of e_k and c_{k+1} is called the *right subtree* of e_k.

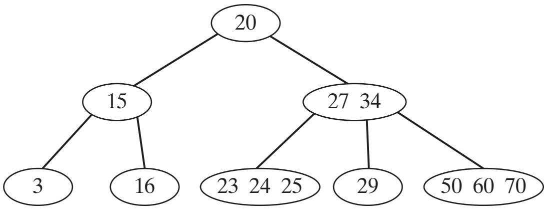


FIGURE 42.2 A 2–4 tree is a full complete search tree.

In a binary tree, each node contains one element. A 2–4 tree tends to be shorter than a corresponding binary search tree, since a 2–4 tree node may contain two or three elements.

Pedagogical Note

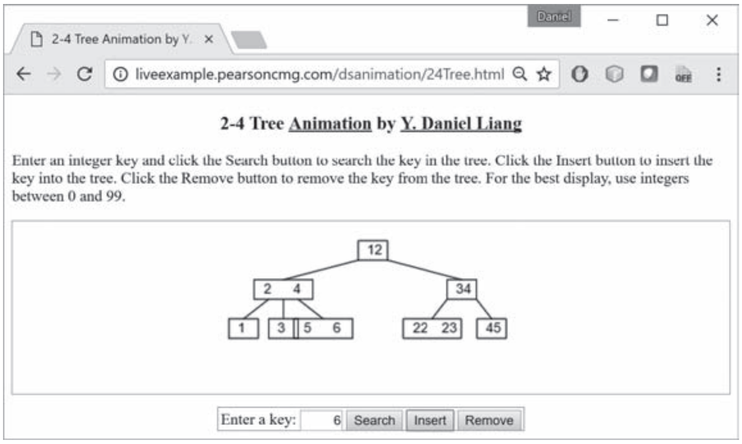
Run from http://liveexample.pearsoncmg.com/dsanimation/24Tree.html to see how a 2–4 tree works, as shown in Figure 42.3.

42.2 Designing Classes for 2–4 Trees

Key Point

The `Tree24` class defines a 2–4 tree and provides methods for searching, inserting, and deleting elements.

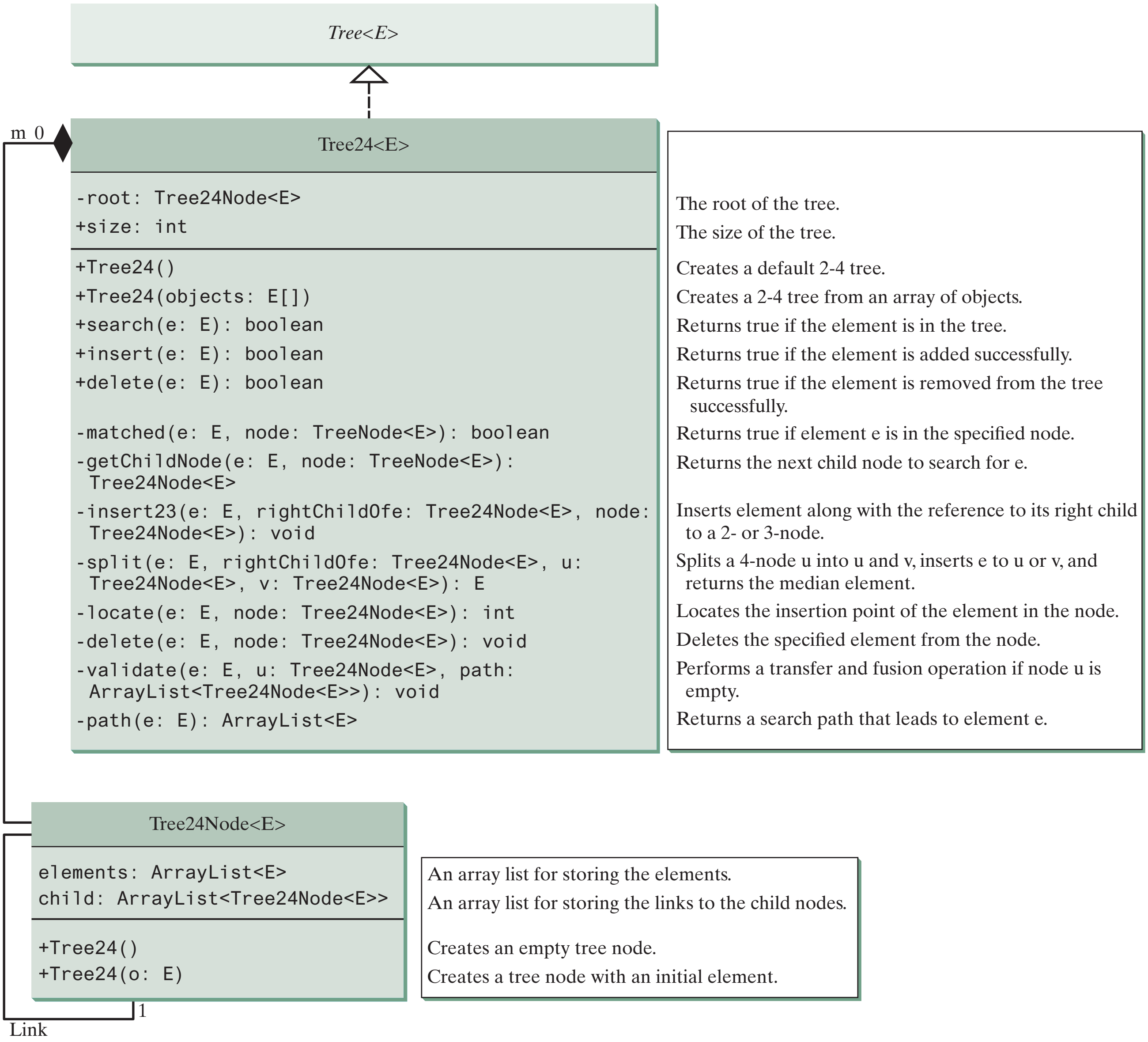
The `Tree24` class can be designed by implementing the `Tree` interface, as shown in Figure 42.4. The `Tree` interface was defined in Listing 27.3 Tree.java. The `Tree24Node` class defines tree nodes. The elements in the node are stored in a list named `elements` and the links to the child nodes are stored in a list named `child`, as shown in Figure 42.5.



2-4 Tree Animation by Y. Daniel Liang

Enter an integer key and click the Search button to search the key in the tree. Click the Insert button to insert the key into the tree. Click the Remove button to remove the key from the tree. For the best display, use integers between 0 and 99.

FIGURE 42.3 The animation tool enables you to insert, delete, and search elements in a 2–4 tree visually.



Tree24<E>	
`-root: Tree24Node<E>`	The root of the tree.
`+size: int`	The size of the tree.
`+Tree24()`	Creates a default 2-4 tree.
`+Tree24(objects: E[])`	Creates a 2-4 tree from an array of objects.
`+search(e: E): boolean`	Returns true if the element is in the tree.
`+insert(e: E): boolean`	Returns true if the element is added successfully.
`+delete(e: E): boolean`	Returns true if the element is removed from the tree successfully.
`-matched(e: E, node: TreeNode<E>): boolean`	Returns true if element e is in the specified node.
`-getChildNode(e: E, node: TreeNode<E>): Tree24Node<E>`	Returns the next child node to search for e.
`-insert23(e: E, rightChildOfe: Tree24Node<E>, node: Tree24Node<E>): void`	Inserts element along with the reference to its right child to a 2- or 3-node.
`-split(e: E, rightChildOfe: Tree24Node<E>, u: Tree24Node<E>, v: Tree24Node<E>): E`	Splits a 4-node u into u and v, inserts e to u or v, and returns the median element.
`-locate(e: E, node: Tree24Node<E>): int`	Locates the insertion point of the element in the node.
`-delete(e: E, node: Tree24Node<E>): void`	Deletes the specified element from the node.
`-validate(e: E, u: Tree24Node<E>, path: ArrayList<Tree24Node<E>>): void`	Performs a transfer and fusion operation if node u is empty.
`-path(e: E): ArrayList<E>`	Returns a search path that leads to element e.

Tree24Node<E>	
`elements: ArrayList<E>`	An array list for storing the elements.
`child: ArrayList<Tree24Node<E>>`	An array list for storing the links to the child nodes.
`+Tree24()`	Creates an empty tree node.
`+Tree24(o: E)`	Creates a tree node with an initial element.

FIGURE 42.4 The `Tree24` class implements `Tree`.

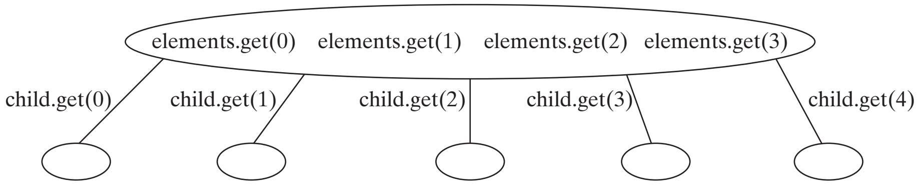


FIGURE 42.5 A 2–4 tree node stores the elements and the links to the child nodes in array lists.

Check Point

42.2.1 What is a 2–4 tree? What are a 2-node, 3-node, and 4-node?

42.2.2 Describe the data fields in the `Tree24` class and those in the `Tree24Node` class.

42.2.3 What is the minimum number of elements in a 2–4 tree of height 5? What is the maximum number of elements in a 2–4 tree of height 5?

42.3 Searching an Element

Key Point

Searching an element in a 2–4 tree is similar to searching an element in a binary tree. The difference is that you have to search an element within a node in addition to searching elements along the path.

To search an element in a 2–4 tree, you start from the root and scan down. If an element is not in the node, move to an appropriate subtree. Repeat the process until a match is found or you arrive at an empty subtree. The algorithm is described in Listing 42.1.

LISTING 42.1 Searching an Element in a 2–4 tree

```
 1  boolean search(E e) {
 2    current = root; // Start from the root
 3
 4    while (current != null) {
 5      if (match(e, current)) { // Element is in the node
 6        return true; // Element is found
 7      }
 8      else {
 9        current = getChildNode(e, current); // Search in a subtree
10      }
11    }
12    return false; // Element is not in the tree
13  }
```

The `match(e, current)` method checks whether element `e` is in the current node. The `getChildNode(e, current)` method returns the root of the subtree for further search. Initially, let `current` point to the root (line 2). Repeat searching the element in the current node until `current` is `null` (line 4) or the element matches an element in the current node.

42.4 Inserting an Element into a 2–4 tree

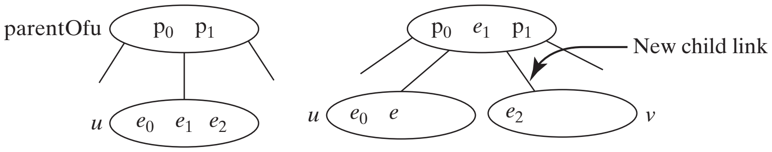
Key Point

Inserting an element involves locating a leaf node and inserting the element into the leaf node.

To insert an element *e* to a 2–4 tree, locate a leaf node in which the element will be inserted. If the leaf node is a 2-node or 3-node, simply insert the element into the node. If the node is a 4-node, inserting a new element would cause an *overflow*. To resolve overflow, perform a *split* operation as follows:

- Let *u* be the *leaf* 4-node in which the element will be inserted and *parentOfu* be the parent of *u*, as shown in Figure 42.6(a).

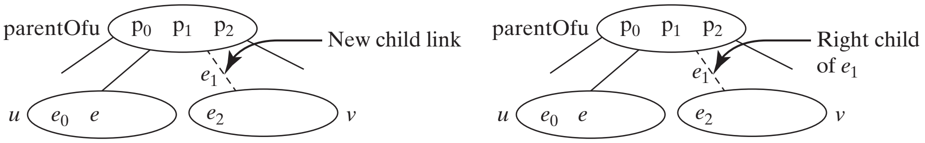
- Create a new node named *v*; move e_2 to *v*.
- If $e < e_1$, insert *e* to *u*; otherwise insert *e* to *v*. Assume that $e_0 < e < e_1$, *e* is inserted into *u*, as shown in Figure 42.6(b).
- Insert e_1 along with its right child (i.e., *v*) to the parent node, as shown in Figure 42.6(b).



(a) Before inserting *e* (b) After inserting *e*

FIGURE 42.6 The splitting operation creates a new node and inserts the median element to its parent.

The parent node is a 3-node in Figure 42.6. So, there is room to insert *e* to the parent node. What happens if it is a 4-node, as shown in Figure 42.7? This requires that the parent node be split. The process is the same as splitting a leaf 4-node, except that you must also insert the element along with its right child.



(a) The parent is a 4-node (b) Inserting e_1 into the parent

FIGURE 42.7 Insertion process continues if the parent node is a 4-node.

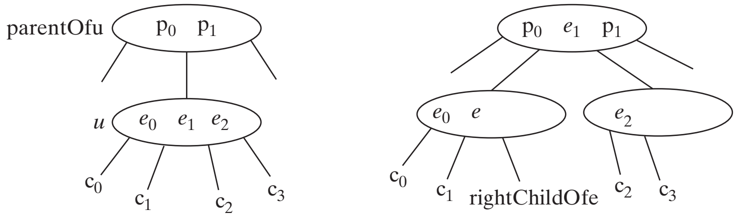
The algorithm can be modified as follows:

- Let *u* be the 4-node (*leaf or nonleaf*) in which the element will be inserted and *parentOfu* be the parent of *u*, as shown in Figure 42.8(a).
- Create a new node named *v*, move e_2 and its children c_2 and c_3 to *v*.
- If $e < e_1$, insert *e* along with its right child link to *u*; otherwise insert *e* along with its right child link to *v*, as shown in Figure 42.6(b), (c), (d) for the cases $e_0 < e < e_1$, $e_1 < e < e_2$, and $e_2 < e$, respectively.
- Insert e_1 along with its right child (i.e., *v*) to the parent node, recursively.

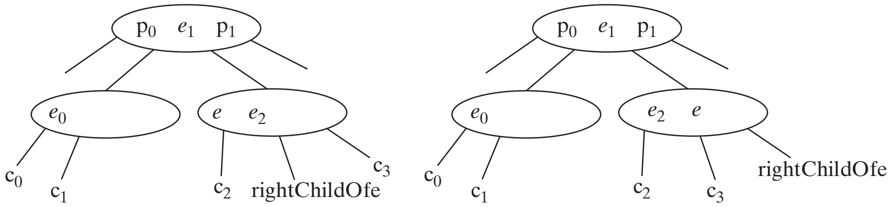
Listing 42.2 gives an algorithm for inserting an element.

LISTING 42.2 Inserting an Element to a 2-4 tree

```
public boolean insert(E e) {
  if (root == null)
    root = new Tree24Node<E>(e); // Create a new root for element
  else {
    Locate leafNode for inserting e
    insert(e, null, leafNode); // The right child of e is null
  }

  size++; // Increase size
  return true; // Element inserted
}
```



(a) Before inserting *e* (b) After inserting *e* $(e_0 < e < e_1)$



(c) After inserting *e* $(e_1 < e < e_2)$ (d) After inserting *e* $(e_2 < e)$

FIGURE 42.8 An interior node may be split to resolve overflow.

```
13  private void insert(E e, Tree24Node<E> rightChildOfe,
14      Tree24Node<E> u) {
15    if (u is a 2- or 3- node) { // u is a 2- or 3-node
16      insert23(e, rightChildOfe, u); // Insert e to node u
17    }
18    else { // Split a 4-node u
19      Tree24Node<E> v = new Tree24Node<E>(); // Create a new node
20      E median = split(e, rightChildOfe, u, v); // Split u
21
22      if (u == root) { // u is the root
23        root = new Tree24Node<E>(median); // New root
24        root.child.add(u); // u is the left child of median
25        root.child.add(v); // v is the right child of median
26      }
27      else {
28        Get the parent of u, parentOfu;
29        insert(median, v, parentOfu); // Inserting median to parent
30      }
31    }
32  }
```

The `insert(E e, Tree24Node<E> rightChildOfe, Tree24Node<E> u)` method inserts element `e` along with its right child to node `u`. When inserting `e` to a leaf node, the right child of `e` is `null` (line 6). If the node is a 2- or 3-node, simply insert the element to the node (lines 15–17). If the node is a 4-node, invoke the `split` method to split the node (line 20). The `split` method returns the median element. Recursively invoke the `insert` method to insert the median element to the parent node (line 29). Figure 42.9 shows the steps of inserting elements `34`, `3`, `50`, `20`, `15`, `16`, `25`, `27`, `29`, and `24` into a 2–4 tree.

42.5 Deleting an Element from a 2–4 tree

Key Point

Deleting an element involves locating the node that contains the element and removing the element from the node.

To delete an element from a 2–4 tree, first search the element in the tree to locate the node that contains it. If the element is not in the tree, the method returns false. Let *u* be the node that contains the element and *parentOfu* be the parent of *u*. Consider three cases:

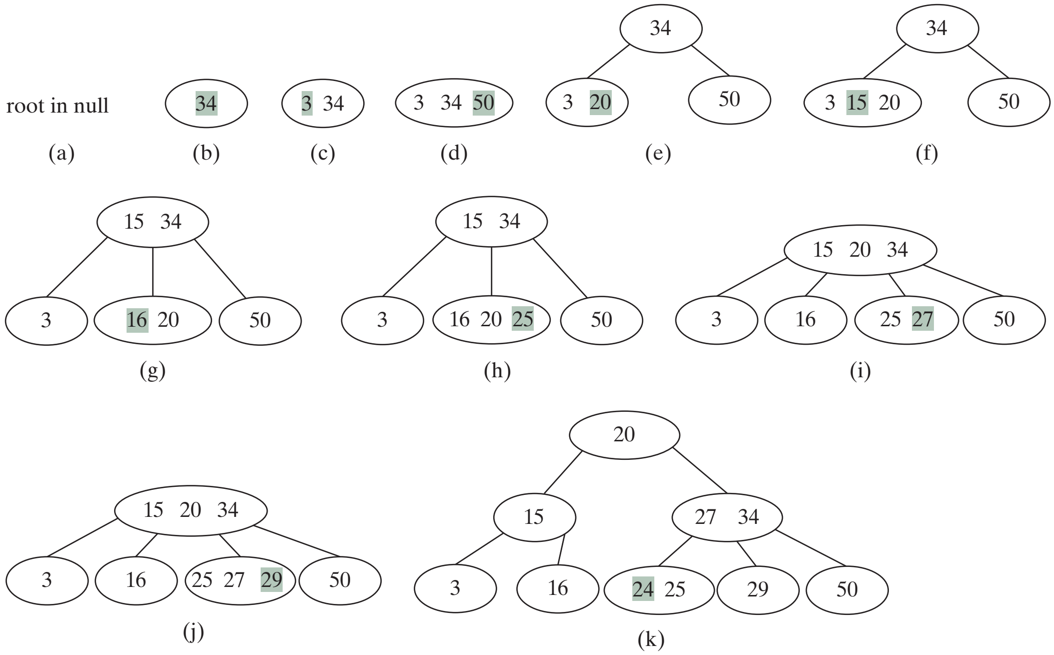
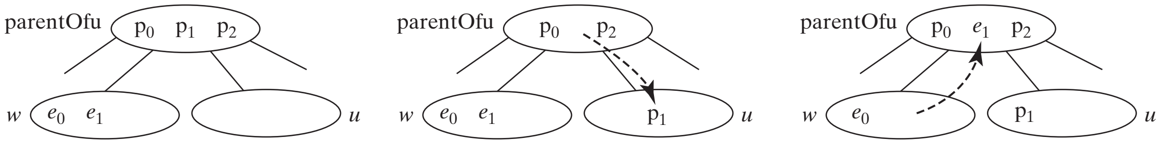

root in null
(a)
34
(b)
3 34
(c)
3 34 50
(d)
34
3 20
50
(e)
34
3 15 20
50
(f)
15 34
3
16 20
50
(g)
15 34
3
16 20 25
50
(h)
15 20 34
3
16
25 27
50
(i)
15 20 34
3
16
25 27 29
50
(j)
20
15
27 34
3
16
24 25
29
50
(k)

FIGURE 42.9 The tree changes after **34**, **3**, **50**, **20**, **15**, **16**, **25**, **27**, **29**, and **24** are added into an empty tree.

Case 1: *u* is a leaf 3-node or 4-node. Delete *e* from *u*.

Case 2: *u* is a leaf 2-node. Delete *e* from *u*. Now *u* is empty. This situation is known as *underflow*. To remedy an underflow, consider two subcases:

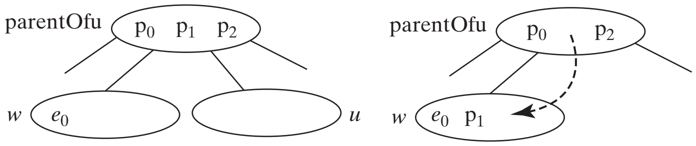
Case 2.1: *u*'s immediate left or right sibling is a 3- or 4-node. Let the node be *w*, as shown in Figure 42.10(a) (assume that *w* is a left sibling of *u*). Perform a *transfer* operation that moves an element from *parentOfu* to *u*, as shown in Figure 42.10(b), and move an element from *w* to replace the moved element in *parentOfu*, as shown in Figure 42.10(c).


parentOfu
p_0 p_1 p_2
w
e_0 e_1
u
parentOfu
p_0 p_2
w
e_0 e_1
p_1
u
parentOfu
p_0 e_1 p_2
w
e_0
p_1
u

(a) *u* is now empty (b) Move p_1 to *u* (c) Move e_1 to replace p_1

FIGURE 42.10 The transfer operation fills the empty node *u*.

Case 2.2: Both *u*'s immediate left and right siblings are 2-node if they exist (*u* may have only one sibling). Let the node be *w*, as shown in Figure 42.11(a) (assume that *w* is a left sibling of *u*). Perform a *fusion* operation that discards *u* and moves an element from *parentOfu* to *w*, as shown in Figure 42.11(b). If *parentOfu* becomes empty, repeat Case 2 recursively to perform a transfer or a fusion on *parentOfu*.

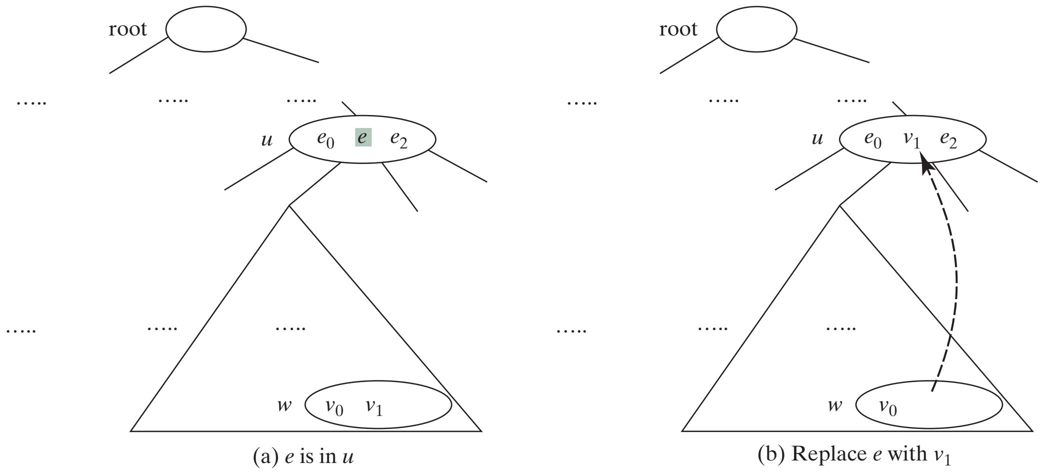

parentOfu
p_0 p_1 p_2
w
e_0
u
parentOfu
p_0 p_2
w
e_0 p_1

(a) *w* is a 2-node (b) Move p_1 to *w*

FIGURE 42.11 The fusion operation discards the empty node *u*.

Case 3: *u* is a nonleaf node. Find the rightmost leaf node in the left subtree of *e*. Let this node be *w*, as shown in Figure 42.12(a). Move the last element in *w* to replace *e* in *u*, as shown in Figure 42.12(b). If *w* becomes empty, apply a transfer or fusion operation on *w*.

Listing 42.3 describes the algorithm for deleting an element.



(a) *e* is in *u*

(b) Replace *e* with v_1

FIGURE 42.12 An element in the internal node is replaced by an element in a leaf node.

LISTING 42.3 Deleting an Element from a 2–4 tree

```
/** Delete the specified element from the tree */
public boolean delete(E e) {
  Locate the node that contains the element e
  if (the node is found) {
    delete(e, node); // Delete element e from the node
    size--; // After one element deleted
    return true; // Element deleted successfully
  }

  return false; // Element not in the tree
}

/** Delete the specified element from the node */
private void delete(E e, Tree24Node<E> node) {
  if (e is in a leaf node) {
    // Get the path that leads to e from the root
    ArrayList<Tree24Node<E>> path = path(e);

    Remove e from the node;

    // Check node for underflow along the path and fix it
    validate(e, node, path); // Check underflow node
  }
  else { // e is in an internal node
    Locate the rightmost node in the left subtree of node u;
    Get the rightmost element from the rightmost node;

    // Get the path that leads to e from the root
    ArrayList<Tree24Node<E>> path = path(rightmostElement);

    Replace the element in the node with the rightmost element

33      // Check node for underflow along the path and fix it
34      validate(rightmostElement, rightmostNode, path);
35    }
36  }
37
38  /** Perform a transfer or fusion operation if necessary */
39  private void validate(E e, Tree24Node<E> u,
40      ArrayList<Tree24Node<E>> path) {
41    for (int i = path.size() - 1; i >= 0; i--) {
42      if (u is not empty)
43        return; // Done, no need to perform transfer or fusion
44
45      Tree24Node<E> parentOfu = path.get(i - 1); // Get parent of u
46
47      // Check two siblings
48      if (left sibling of u has more than one element) {
49        Perform a transfer on u with its left sibling
50      }
51      else if (right sibling of u has more than one element) {
52        Perform a transfer on u with its right sibling
53      }
54      else if (u has left sibling) { // Fusion with a left sibling
55        Perform a fusion on u with its left sibling
56        u = parentOfu; // Back to the loop to check the parent node
57      }
58      else { // Fusion with right sibling (right sibling must exist)
59        Perform a fusion on u with its right sibling
60        u = parentOfu; // Back to the loop to check the parent node
61      }
62    }
63  }
```

The `delete(E e)` method locates the node that contains the element `e` and invokes the `delete(E e, Tree24Node<E> node)` method (line 5) to delete the element from the node.

If the node is a leaf node, get the path that leads to `e` from the root (line 17), delete `e` from the node (line 19), and invoke `validate` to check and fix the empty node (line 22). The `validate(E e, Tree24Node<E> u, ArrayList<Tree24Node<E>> path)` method performs a transfer or fusion operation if the node is empty. Since these operations may cause the parent of node `u` to become empty, a path is obtained in order to obtain the parents along the path from the root to node `u`, as shown in Figure 42.13.

If the node is a nonleaf node, locate the rightmost element in the left subtree of the node (lines 25–26), get the path that leads to the rightmost element from the root (line 29), replace

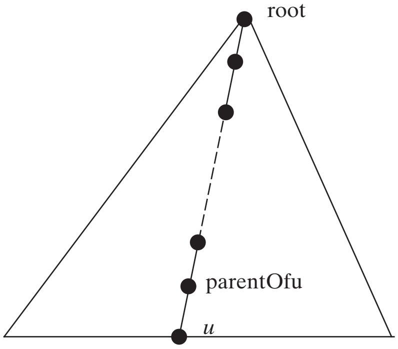


Figure 42.13 The nodes along the path may become empty as result of a transfer and fusion operation.

`e` in the node with the rightmost element (line 31), and invoke `validate` to fix the rightmost node if it is empty (line 34).

The `validate(E e, Tree24Node<E> u, ArrayList<Tree24Node<E>> path)` checks whether `u` is empty and performs a transfer or fusion operation to fix the empty node. The `validate` method exits when node is not empty (line 43). Otherwise, consider one of the following cases:

1. If `u` has a left sibling with more than one element, perform a transfer on `u` with its left sibling (line 49).
2. Otherwise, if `u` has a right sibling with more than one element, perform a transfer on `u` with its right sibling (line 52).
3. Otherwise, if `u` has a left sibling, perform a fusion on `u` with its left sibling (line 55) and reset `u` to `parentOfu` (line 56).
4. Otherwise, `u` must have a right sibling. Perform a fusion on `u` with its right sibling (line 59) and reset `u` to `parentOfu` (line 60).

Only one of the preceding cases is executed. Afterward, a new iteration starts to perform a transfer or fusion operation on a new node `u` if needed. Figure 42.14 shows the steps of deleting elements `20`, `15`, `3`, `6`, and `34` that are deleted from a 2–4 tree in Figure 42.9(k).

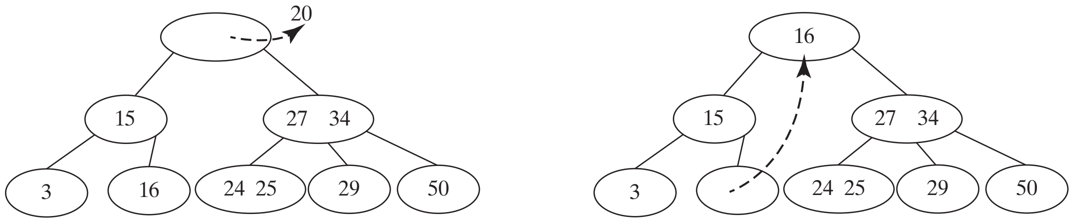
Check Point

42.5.1 How do you search an element in a 2–4 tree?

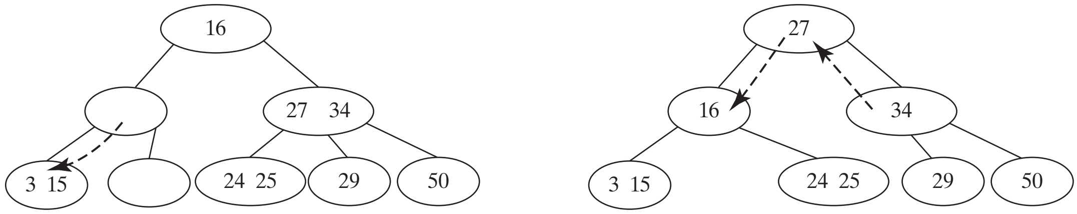
42.5.2 How do you insert an element into a 2–4 tree?

42.5.3 How do you delete an element from a 2–4 tree?

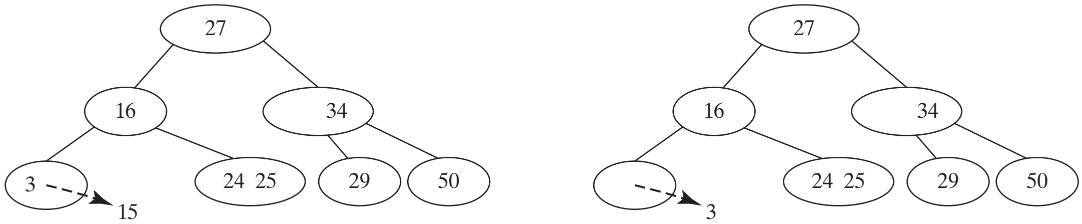
42.5.4 Show the change of a 2–4 tree when inserting `1`, `2`, `3`, `4`, `10`, `9`, `7`, `5`, `8`, and `6` into it, in this order.



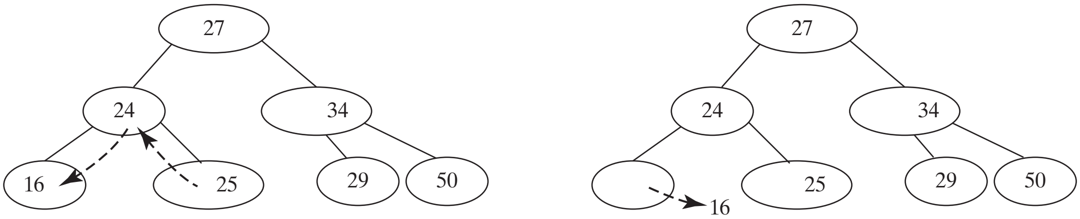
(a) Delete 20 (b) Replace 20 with 16



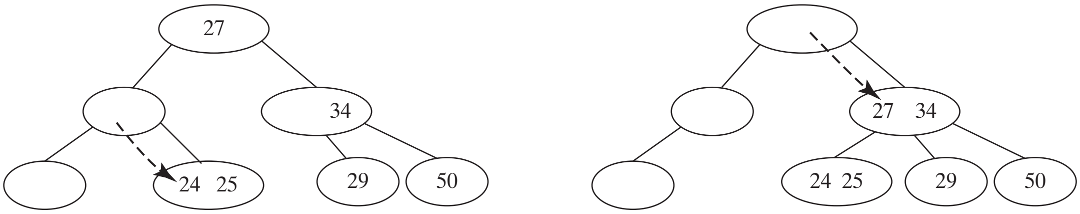
(c) Perform a fusion (d) Perform a transfer



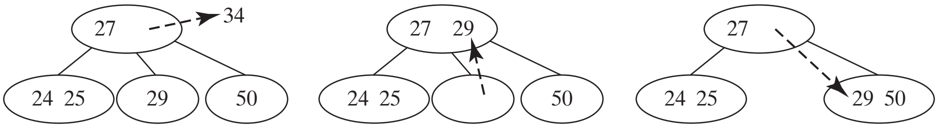
(e) Delete 15 (f) Delete 3



(g) Perform a transfer (h) Delete 16



(i) Perform a fusion (j) Perform a fusion



(k) Delete 34 (l) Replace 34 with 16 (m) Perform a fusion

Figure 42.14 The tree changes after **20**, **15**, **3**, **6**, and **34** are deleted from a 2–4 tree.

42.5.5 For the tree built in the preceding question, show the change of the tree after deleting **1**, **2**, **3**, **4**, **10**, **9**, **7**, **5**, **8**, and **6** from it, in this order.

42.5.6 Show the change of a B-tree of order 6 when inserting **1**, **2**, **3**, **4**, **10**, **9**, **7**, **5**, **8**, **6**, **17**, **25**, **18**, **26**, **14**, **52**, **63**, **74**, **80**, **19**, and **27** into it, in this order.

42.5.7 For the tree built in the preceding question, show the change of the tree after deleting **1**, **2**, **3**, **4**, **10**, **9**, **7**, **5**, and **8**, and **6** from it, in this order.

42.6 Traversing Elements in a 2–4 tree

You can perform inorder, preorder, and postorder for traversing the elements in a 2–4 tree.

Inorder, preorder, and postorder traversals are useful for 2–4 trees. Inorder traversal visits the elements in increasing order. Preorder traversal visits the elements in the root, then recursively visits the subtrees from the left to right. Postorder traversal visits the subtrees from the left to right recursively, and then the elements in the root.

For example, in the 2–4 tree in Figure 42.9(k), the inorder traversal is
3 15 16 20 24 25 27 29 34 50

The preorder traversal is
20 15 3 16 27 34 24 25 29 50

The postorder traversal is
3 16 1 24 25 29 50 27 34 20

42.7 Implementing the `Tree24` Class

*This section gives the complete implementation for the **`Tree24`** class.*

Listing 42.4 gives the complete source code for the `Tree24` class.

LISTING 42.4 Tree24.java

```
import java.util.ArrayList;

public class Tree24<E extends Comparable<E>> implements Tree<E> {
  private Tree24Node<E> root;
  private int size;

  /** Create a default 2-4 tree */
  public Tree24() {
  }

  /** Create a 2-4 tree from an array of objects */
  public Tree24(E[] elements) {
    for (int i = 0; i < elements.length; i++)
      insert(elements[i]);
  }

  @Override /* Search an element in the tree */
  public boolean search(E e) {
    Tree24Node<E> current = root; // Start from the root

    while (current != null) {
      if (matched(e, current)) { // Element is in the node
        return true; // Element found
      }
      else {
        current = getChildNode(e, current); // Search in a subtree
      }
    }

    return false; // Element is not in the tree
  }

  /** Return true if the element is found in this node */
  private boolean matched(E e, Tree24Node<E> node) {
    for (int i = 0; i < node.elements.size(); i++)
      if (node.elements.get(i).equals(e))
        return true; // Element found

    return false; // No match in this node
  }

  /** Locate a child node to search element e */
  private Tree24Node<E> getChildNode(E e, Tree24Node<E> node) {
    if (node.child.size() == 0)
      return null; // node is a leaf

    int i = locate(e, node); // Locate the insertion point for e
    return node.child.get(i); // Return the child node
  }

  @Override /** Insert element e into the tree
   * Return true if the element is inserted successfully
   */
  public boolean insert(E e) {
    if (root == null)
      root = new Tree24Node<E>(e); // Create a new root for element
    else {
      // Locate the leaf node for inserting e
      Tree24Node<E> leafNode = null;
      Tree24Node<E> current = root;
      while (current != null)
        if (matched(e, current)) {
          return false; // Duplicate element found, nothing inserted
        }
        else {
          leafNode = current;
          current = getChildNode(e, current);
        }

      // Insert the element e into the leaf node
      insert(e, null, leafNode); // The right child of e is null
    }

    size++; // Increase size
    return true; // Element inserted
  }

  /** Insert element e into node u */
  private void insert(E e, Tree24Node<E> rightChildOfe,
      Tree24Node<E> u) {
    // Get the search path that leads to element e
    ArrayList<Tree24Node<E>> path = path(e);

    for (int i = path.size() - 1; i >= 0; i--) {
      if (u.elements.size() < 3) { // u is a 2-node or 3-node
        insert23(e, rightChildOfe, u); // Insert e to node u
        break; // No further insertion to u's parent needed
      }
      else {
        Tree24Node<E> v = new Tree24Node<E>(); // Create a new node
        E median = split(e, rightChildOfe, u, v); // Split u

        if (u == root) {
          root = new Tree24Node<E>(median); // New root
          root.child.add(u); // u is the left child of median
          root.child.add(v); // v is the right child of median
          break; // No further insertion to u's parent needed
        }
        else {
          // Use new values for the next iteration in the for loop
          e = median; // Element to be inserted to parent
          rightChildOfe = v; // Right child of the element
          u = path.get(i - 1); // New node to insert element
        }
      }
    }
  }

  /** Insert element to a 2- or 3- and return the insertion point */
  private void insert23(E e, Tree24Node<E> rightChildOfe,
      Tree24Node<E> node) {
    int i = this.locate(e, node); // Locate where to insert
    node.elements.add(i, e); // Insert the element into the node
    if (rightChildOfe != null)
      node.child.add(i + 1, rightChildOfe); // Insert the child link
  }

  /** Split a 4-node u into u and v and insert e to u or v */
  private E split(E e, Tree24Node<E> rightChildOfe,
      Tree24Node<E> u, Tree24Node<E> v) {
    // Move the last element in node u to node v
    v.elements.add(u.elements.remove(2));
    E median = u.elements.remove(1);

    // Split children for a nonleaf node
    // Move the last two children in node u to node v
    if (u.child.size() = 0) {
      v.child.add(u.child.remove(2));
      v.child.add(u.child.remove(2));
    }

    // Insert e into a 2- or 3- node u or v.
    if (e.compareTo(median) < 0)
      insert23(e, rightChildOfe, u);
    else
      insert23(e, rightChildOfe, v);

    return median; // Return the median element
  }

  /** Return a search path that leads to element e */
  private ArrayList<Tree24Node<E>= path(E e) {
    ArrayList<Tree24Node<E>= list = new ArrayList<Tree24Node<E>=();
    Tree24Node<E> current = root; // Start from the root

    while (current != null) {
      list.add(current); // Add the node to the list
      if (matched(e, current)) {
        break; // Element found
      }
      else {
        current = getChildNode(e, current);
      }
    }

    return list; // Return an array of nodes
  }

  @Override /** Delete the specified element from the tree */
  public boolean delete(E e) {
    // Locate the node that contains the element e
    Tree24Node<E> node = root;
    while (node != null)
      if (matched(e, node)) {
        delete(e, node); // Delete element e from node
        size--; // After one element deleted
        return true; // Element deleted successfully
      }
      else {
        node = getChildNode(e, node);
      }

    return false; // Element not in the tree
  }

  /** Delete the specified element from the node */
  private void delete(E e, Tree24Node<E> node) {
    if (node.child.size() == 0) { // e is in a leaf node
      // Get the path that leads to e from the root
      ArrayList<Tree24Node<E>> path = path(e);
```

```

        node.elements.remove(e); // Remove element e

        if (node == root) { // Special case
          if (node.elements.size() == 0)
            root = null; // Empty tree
          return; // Done
        }

        validate(e, node, path); // Check underflow node
      }
      else { // e is in an internal node
        // Locate the rightmost node in the left subtree of the node
        int index = locate(e, node); // Index of e in node
        Tree24Node<E> current = node.child.get(index);
        while (current.child.size() > 0) {
          current = current.child.get(current.child.size() - 1);
        }
        E rightmostElement =
          current.elements.get(current.elements.size() - 1);

        // Get the path that leads to e from the root
        ArrayList<Tree24Node<E>= path = path(rightmostElement);

        // Replace the deleted element with the rightmost element
        node.elements.set(index, current.elements.remove(
          current.elements.size() - 1));

        validate(rightmostElement, current, path); // Check underflow
      }
    }

    /** Perform transfer and confusion operations if necessary */
    private void validate(E e, Tree24Node<E> u,
        ArrayList<Tree24Node<E>> path) {
      for (int i = path.size() - 1; u.elements.size() == 0; i--) {
        Tree24Node<E> parentOfu = path.get(i - 1); // Get parent of u
        int k = locate(e, parentOfu); // Index of e in the parent node

        // Check two siblings
        if (k > 0 && parentOfu.child.get(k - 1).elements.size() > 1) {
          leftSiblingTransfer(k, u, parentOfu);
        }
        else if (k + 1 < parentOfu.child.size() &&
            parentOfu.child.get(k + 1).elements.size() > 1) {
          rightSiblingTransfer(k, u, parentOfu);
        }
        else if (k - 1 == 0) { // Fusion with a left sibling
          // Get left sibling of node u
          Tree24Node<E> leftNode = parentOfu.child.get(k - 1);

          // Perform a fusion with left sibling on node u
          leftSiblingFusion(k, leftNode, u, parentOfu);

          // Done when root becomes empty
          if (parentOfu == root && parentOfu.elements.size() == 0) {
            root = leftNode;
            break;
          }

          u = parentOfu; // Back to the loop to check the parent node
      }
      else { // Fusion with right sibling (right sibling must exist)
        // Get left sibling of node u
        Tree24Node<E> rightNode = parentOfu.child.get(k + 1);

        // Perform a fusion with right sibling on node u
        rightSiblingFusion(k, rightNode, u, parentOfu);

        // Done when root becomes empty
        if (parentOfu == root && parentOfu.elements.size() == 0) {
          root = rightNode;
          break;
        }

        u = parentOfu; // Back to the loop to check the parent node
      }
    }
  }

  /** Locate the insertion point of the element in the node */
  private int locate(E o, Tree24Node<E> node) {
    for (int i = 0; i < node.elements.size(); i++) {
      if (o.compareTo(node.elements.get(i)) <= 0) {
        return i;
      }
    }

    return node.elements.size();
  }

  /** Perform a transfer with a left sibling */
  private void leftSiblingTransfer(int k,
      Tree24Node<E> u, Tree24Node<E> parentOfu) {
    // Move an element from the parent to u
    u.elements.add(0, parentOfu.elements.get(k - 1));

    // Move an element from the left node to the parent
    Tree24Node<E> leftNode = parentOfu.child.get(k - 1);
    parentOfu.elements.set(k - 1,
      leftNode.elements.remove(leftNode.elements.size() - 1));

    // Move the child link from left sibling to the node
    if (leftNode.child.size() > 0)
      u.child.add(0, leftNode.child.remove(
        leftNode.child.size() - 1));
  }

  /** Perform a transfer with a right sibling */
  private void rightSiblingTransfer(int k,
      Tree24Node<E> u, Tree24Node<E> parentOfu) {
    // Transfer an element from the parent to u
    u.elements.add(parentOfu.elements.get(k));

    // Transfer an element from the right node to the parent
    Tree24Node<E> rightNode = parentOfu.child.get(k + 1);
    parentOfu.elements.set(k, rightNode.elements.remove(0));

    // Move the child link from right sibling to the node
    if (rightNode.child.size() > 0)
      u.child.add(rightNode.child.remove(0));
  }
```

```java

  /** Perform a fusion with a left sibling */
  private void leftSiblingFusion(int k, Tree24Node<E> leftNode,
      Tree24Node<E> u, Tree24Node<E> parentOfu) {
    // Transfer an element from the parent to the left sibling
    leftNode.elements.add(parentOfu.elements.remove(k - 1));

    // Remove the link to the empty node
    parentOfu.child.remove(k);

    // Adjust child links for nonleaf node
    if (u.child.size() > 0)
      leftNode.child.add(u.child.remove(0));
  }

  /** Perform a fusion with a right sibling */
  private void rightSiblingFusion(int k, Tree24Node<E> rightNode,
      Tree24Node<E> u, Tree24Node<E> parentOfu) {
    // Transfer an element from the parent to the right sibling
    rightNode.elements.add(0, parentOfu.elements.remove(k));

    // Remove the link to the empty node
    parentOfu.child.remove(k);

    // Adjust child links for nonleaf node
    if (u.child.size() > 0)
      rightNode.child.add(0, u.child.remove(0));
  }

  /** Get the number of nodes in the tree */
  public int getSize() {
    return size;
  }

  /** Preorder traversal from the root */
  public void preorder() {
    preorder(root);
  }

  /** Preorder traversal from a subtree */
  private void preorder(Tree24Node<E> root) {
    if (root == null)return;
    for (int i = 0; i < root.elements.size(); i++)
      System.out.print(root.elements.get(i) + " ");

    for (int i = 0; i < root.child.size(); i++)
      preorder(root.child.get(i));
  }

  /** Inorder traversal from the root*/
  public void inorder() {
    // Left as exercise
  }

  /** Postorder traversal from the root */
  public void postorder() {
    // Left as exercise
  }

  /** Return true if the tree is empty */
  public boolean isEmpty() {
```

```
364       return root == null;
365     }
366 
367     @Override /** Remove all elements from the tree */
368     public void clear() {
369       root = null;
370       size = 0;
371     }
372 
373     /** Return an iterator to traverse elements in the tree */
374     public java.util.Iterator iterator() {
375       // Left as exercise
376       return null;
377     }
378 
379     /** Define a 2-4 tree node */
380     protected static class Tree24Node<E extends Comparable<E>> {
381       // elements has maximum three values
382       ArrayList<E> elements = new ArrayList<E>(3);
383       // Each has maximum four childres
384       ArrayList<Tree24Node<E>> child
385         = new ArrayList<Tree24Node<E>>(4);
386 
387       /** Create an empty Tree24 node */
388       Tree24Node() {
389       }
390 
391       /** Create a Tree24 node with an initial element */
392       Tree24Node(E o) {
393         elements.add(o);
394       }
395     }
396   }
```

The `Tree24` class contains the data fields `root` and `size` (lines 4–5). `root` references the root node and `size` stores the number of elements in the tree.

The `Tree24` class has two constructors: a no-arg constructor (lines 8–9) that constructs an empty tree and a constructor that creates an initial `Tree24` from an array of elements (lines 12–15).

The `search` method (lines 18–31) searches an element in the tree. It returns `true` (line 23) if the element is in the tree and returns `false` if the search arrives at an empty subtree (line 30).

The `matched(e, node)` method (lines 34–40) checks where the element `e` is in the node.

The `getChildNode(e, node)` method (lines 43–49) returns the root of a subtree where `e` should be searched.

The `insert(E e)` method inserts an element in a tree (lines 54–76). If the tree is empty, a new root is created (line 56). The method locates a leaf node in which the element will be inserted and invokes `insert(e, null, leafNode)` to insert the element (line 71).

The `insert(e, rightChildOfe, u)` method inserts an element into node `u` (lines 79–107). The method first invokes `path(e)` (line 82) to obtain a search path from the root to node `u`. Each iteration of the `for` loop considers `u` and its parent `parentOfu` (lines 84–106). If `u` is a 2-node or 3-node, invoke `insert23(e, rightChildOfe, u)` to insert `e` and its child link `rightChildOfe` into `u` (line 86). No split is needed (line 87). Otherwise, create a new node `v` (line 90) and invoke `split(e, rightChildOfe, u, v)` (line 91) to split `u` into `u` and `v`. The `split` method inserts `e` into either `u` and `v` and returns the median in the original `u`. If `u` is the root, create a new root to hold median, and set `u` and `v` as the left and right children for median (lines 95–96). If `u` is not the root, insert median to `parentOfu` in the next iteration (lines 101–103).

The `insert23(e, rightChildOfe, node)` method inserts `e` along with the reference to its right child into the node (lines 110–116). The method first invokes `locate(e, node)` (line 112) to locate an insertion point, then insert `e` into the node (line 113). If `rightChildOfe` is not `null`, it is inserted into the child list of the node (line 115).

The `split(e, rightChildOfe, u, v)` method splits a 4-node `u` (lines 119-139). This is accomplished as follows: (1) move the last element from `u` to `v` and remove the median element from `u` (lines 122–123); (2) move the last two child links from `u` to `v` (lines 127–130) if `u` is a nonleaf node; (3) if `e < median`, insert `e` into `u`; otherwise, insert `e` into `v` (lines 133–136); and (4) return median (line 138).

The `path(e)` method returns an `ArrayList` of nodes searched from the root in order to locate `e` (lines 142–157). If `e` is in the tree, the last node in the path contains `e`. Otherwise, the last node is where `e` should be inserted.

The `delete(E e)` method deletes an element from the tree (lines 160–174). The method first locates the node that contains `e` and invokes `delete(e, node)` to delete `e` from the node (line 165). If the element is not in the tree, return `false` (line 173).

The `delete(e, node)` method deletes an element from node `u` (lines 177–211). If the node is a leaf node, obtain the path that leads to `e` (line 180), delete `e` (line 182), set root to `null` if the tree becomes empty (lines 184–188), and invoke `validate` to apply transfer and fusion operation on empty nodes (line 190). If the node is a nonleaf node, locate the rightmost element (lines 194–200), obtain the path that leads to `e` (line 203), replace `e` with the rightmost element (lines 206–207), and invoke `validate` to apply transfer and fusion operations on empty nodes (line 209).

The `validate(e, u, path)` method ensures that the tree is a valid 2–4 tree (lines 214–259). The `for` loop terminates when `u` is not empty (line 216). The loop body is executed to fix the empty node `u` by performing a transfer or fusion operation. If a left sibling with more than one element exists, perform a transfer on `u` with the left sibling (line 222). Otherwise, if a right sibling with more than one element exists, perform a transfer on `u` with the left sibling (line 226). Otherwise, if a left sibling exists, perform a fusion on `u` with the left sibling (lines 230–239), and validate `parentOfu` in the next loop iteration (line 241). Otherwise, perform a fusion on `u` with the right sibling.

The `locate(e, node)` method locates the index of `e` in the node (lines 262–270).

The `leftSiblingTransfer(k, u, parentOfu)` method performs a transfer on `u` with its left sibling (lines 273–287). The `rightSiblingTransfer(k, u, parentOfu)` method performs a transfer on `u` with its right sibling (lines 290–302). The `leftSiblingFusion(k, leftNode, u, parentOfu)` method performs a fusion on `u` with its left sibling `leftNode` (lines 305–316). The `rightSiblingFusion(k, rightNode, u, parentOfu)` method performs a fusion on `u` with its right sibling `rightNode` (lines 319–330).

The `preorder()` method displays all the elements in the tree in preorder (lines 338–350).

The inner class `Tree24Node` defines a class for a node in the tree (lines 374–389).

42.8 Testing the **Tree24** Class

This section writes a test program for using the ***Tree24*** *class.*

Key Point

Listing 42.5 gives a test program. The program creates a 2–4 tree and inserts elements in lines 6–20, and deletes elements in lines 22–56.

LISTING 42.5 TestTree24.java

```
1  public class TestTree24 {
2    public static void main(String[] args) {
3      // Create a 2-4 tree
4      Tree24<Integer> tree = new Tree24<Integer>();
5
```

```
    tree.insert(34);
    tree.insert(3);
    tree.insert(50);
    tree.insert(20);
    tree.insert(15);
    tree.insert(16);
    tree.insert(25);
    tree.insert(27);
    tree.insert(29);
    tree.insert(24);
    System.out.print("\nAfter inserting 24:");
    printTree(tree);
    tree.insert(23);
    tree.insert(22);
    tree.insert(60);
    tree.insert(70);
    System.out.print("\nAfter inserting 70:");
    printTree(tree);

    tree.delete(34);
    System.out.print("\nAfter deleting 34:");
    printTree(tree);

    tree.delete(25);
    System.out.print("\nAfter deleting 25:");
    printTree(tree);

    tree.delete(50);
    System.out.print("\nAfter deleting 50:");
    printTree(tree);

    tree.delete(16);
    System.out.print("\nAfter deleting 16:");
    printTree(tree);

    tree.delete(3);
    System.out.print("\nAfter deleting 3:");
    printTree(tree);

    tree.delete(15);
    System.out.print("\nAfter deleting 15:");
    printTree(tree);
  }

  public static <E extends Comparable<E>>
    void printTree(Tree<E> tree) {
    // Traverse tree
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```

```
After inserting 24:
Preorder: 20 15 3 16 27 34 24 25 29 50
The number of nodes is 10
```

```
After inserting 70:
Preorder: 20 15 3 16 24 27 34 22 23 25 29 50 60 70
The number of nodes is 14

After deleting 34:
Preorder: 20 15 3 16 24 27 50 22 23 25 29 60 70
The number of nodes is 13

After deleting 25:
Preorder: 20 15 3 16 23 27 50 22 24 29 60 70
The number of nodes is 12

After deleting 50:
Preorder: 20 15 3 16 23 27 60 22 24 29 70
The number of nodes is 11

After deleting 16:
Preorder: 23 20 3 15 22 27 60 24 29 70
The number of nodes is 10

After deleting 3:
Preorder: 23 20 15 22 27 60 24 29 70
The number of nodes is 9

After deleting 15:
Preorder: 27 23 20 22 24 60 29 70
The number of nodes is 8
```

Figure 42.15 shows how the tree evolves as elements are added. After `34`, `3`, `50`, `20`, `15`, `16`, `25`, `27`, `29`, and `24` are added to the tree, it is as shown in Figure 42.15(a). After inserting `23`, `22`, `60`, and `70`, the tree is as shown in Figure 42.15(b). After deleting `34`, the tree is as shown in Figure 42.15(c). After deleting `25`, the tree is as shown in Figure 42.15(d). After deleting `50`, the tree is as shown in Figure 42.15(e). After deleting `16`, the tree is as shown in Figure 42.15(f). After deleting `3`, the tree is as shown in Figure 42.15(g). After deleting `15`, the tree is as shown in Figure 42.15(h).

42.9 Time-Complexity Analysis

Search, insertion, and deletion operations take O(logn) time in a 2–4 tree.

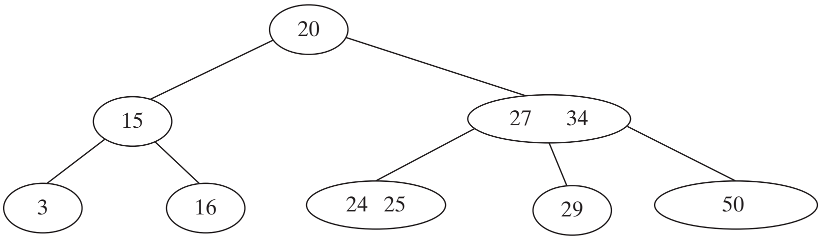
Since a 2–4 tree is a completely balanced binary tree, its height is at most $O(\log n)$. The `search`, `insert`, and `delete` methods operate on the nodes along a path in the tree. It takes a constant time to search an element within a node. So, the `search` method takes $O(\log n)$ time. For the `insert` method, the time for splitting a node takes a constant time. So, the `insert` method takes $O(\log n)$ time. For the `delete` method, it takes a constant time to perform a transfer and fusion operation. So, the `delete` method takes $O(\log n)$ time.

42.10 B-Tree

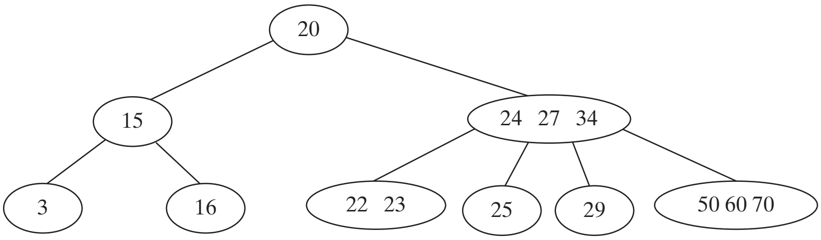
A B-tree is a generalization of a 2–4 tree.

So far we assume that the entire data set is stored in main memory. What if the data set is too large and cannot fit in the main memory, as in the case with most databases, where data is stored on disks? Suppose you use an AVL tree to organize a million records in a database table. To find a record, the average number of nodes traversed is $\log_2 1{,}000{,}000 \approx 20$.

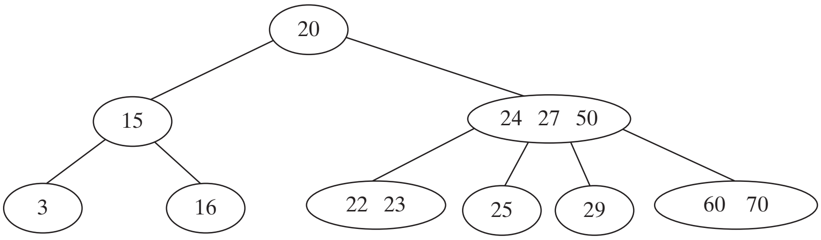
This is fine if all nodes are stored in main memory. However, for nodes stored on a disk, this means 20 disk reads. Disk I/O is expensive, and it is thousands of times slower than memory access. To improve performance, we need to reduce the number of disk I/Os. An efficient data structure for performing search, insertion, and deletion for data stored on secondary storage such as hard disks is the B-tree, which is a generalization of the 2–4 tree.


20
15
27 34
3
16
24 25
29
50

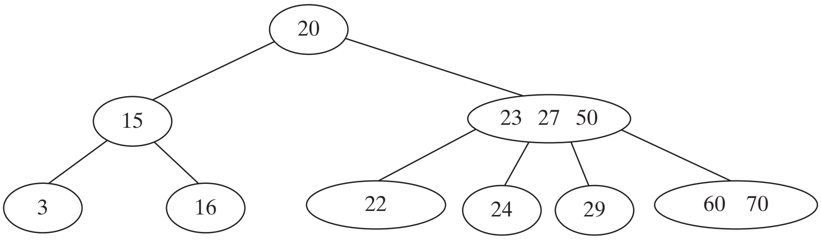
(a) After inserting 34, 3, 50, 20, 15, 16, 25, 27, 29, and 24, in this order


20
15
24 27 34
3
16
22 23
25
29
50 60 70

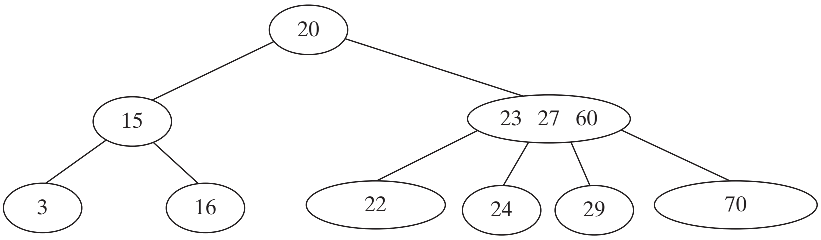
(b) After inserting 23, 22, 60, and 70


20
15
24 27 50
3
16
22 23
25
29
60 70

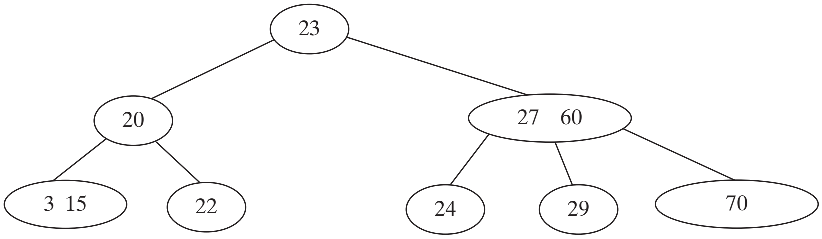
(c) After deleting 34


20
15
23 27 50
3
16
22
24
29
60 70

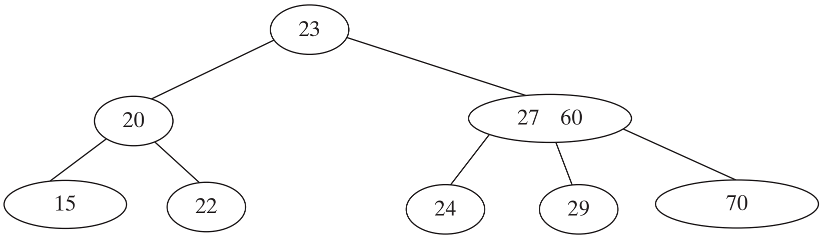
(d) After deleting 25


20
15
23 27 60
3
16
22
24
29
70

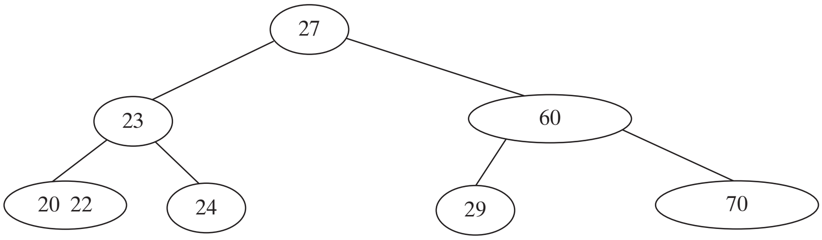
(e) After deleting 50


23
20
27 60
3 15
22
24
29
70

(f) After deleting 16


23
20
27 60
15
22
24
29
70

(g) After deleting 3


27
23
60
20 22
24
29
70

(h) After deleting 15

FIGURE 42.15 The tree evolves as elements are inserted and deleted.

A B-tree of order d is defined as follows:

1. Each node except the root contains between $\lceil d/2 \rceil - 1$ and $d - 1$ keys.
2. The root may contain up to $d - 1$ keys.
3. A nonleaf node with k keys has $k + 1$ children.
4. All leaf nodes have the same depth.

Figure 42.16 shows a B-tree of order 6. For simplicity, we use integers to represent keys. Each key is associated with a pointer that points to the actual record in the database. For simplicity, the pointers to the records in the database are omitted in the figure.

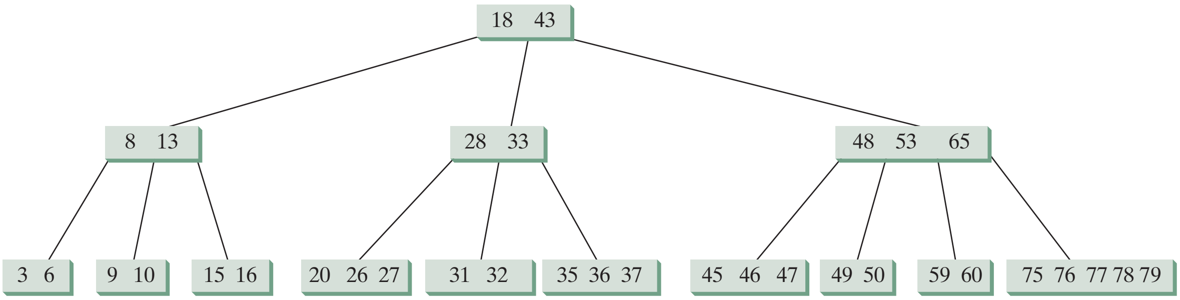

18 43
8 13
28 33
48 53 65
3 6
9 10
15 16
20 26 27
31 32
35 36 37
45 46 47
49 50
59 60
75 76 77 78 79

FIGURE 42.16 In a B-tree of order 6, each node except the root may contain between 2 and 5 keys.

Note that a B-tree is a search tree. The keys in each node are placed in increasing order. Each key in an interior node has a left subtree and a right subtree, as shown in Figure 42.17. All keys in the left subtree are less than the key in the parent node, and all keys in the right subtree are greater than the key in the parent node.

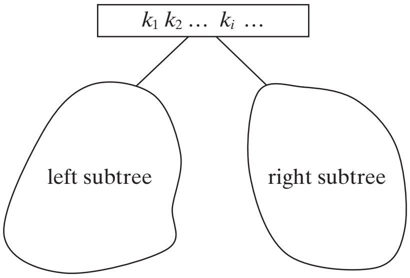


FIGURE 42.17 The keys in the left (right) subtree of key k_i are less than (greater than) k_i.

The basic unit of the IO operations on a disk is a block. When you read data from a disk, the whole block that contains the data is read. You should choose an appropriate order *d* so that a node can fit in a single disk block. This will minimize the number of disk IOs.

A 2–4 tree is actually a B-tree of order 4. The techniques for insertion and deletion in a 2–4 tree can be easily generalized for a B-tree.

Inserting a key to a B-tree is similar to what was done for a 2–4 tree. First locate the leaf node in which the key will be inserted. Insert the key to the node. After the insertion, if the leaf node has *d* keys, an overflow occurs. To resolve overflow, perform a *split* operation similar to the one used in a 2–4 tree, as follows:

Let *u* denote the node needed to be split and let `m` denote the median key in the node. Create a new node and move all keys greater than `m` to this new node. Insert `m` to the parent node of *u*. Now `u` becomes the left child of `m` and `v` becomes the right child of `m`, as shown in Figure 42.18. If inserting `m` into the parent node of `u` causes an overflow, repeat the same split process on the parent node.

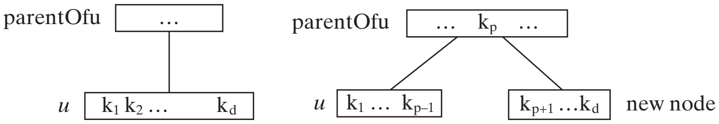
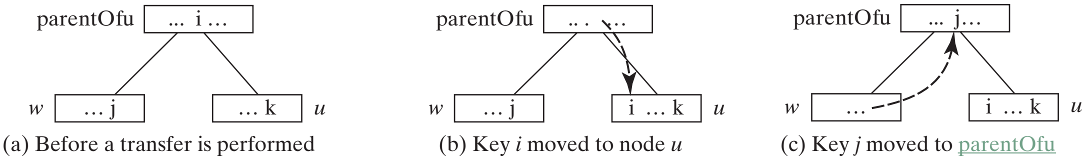


FIGURE 42.18 (a) After inserting a new key to node *u*. (b) The median key k_p is inserted to `parentOfu`.

A key *k* can be deleted from a B-tree in the same way as in a 2–4 tree. First locate the node *u* that contains the key. Consider two cases:

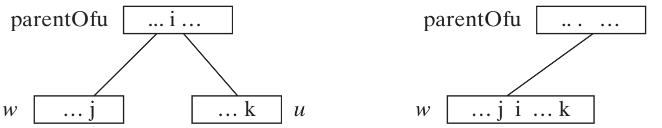
Case 1: If *u* is a leaf node, remove the key from *u*. After the removal, if *u* has less than $\lceil d/2 \rceil - 1$ keys, an underflow occurs. To remedy an underflow, perform a transfer with a sibling



(a) Before a transfer is performed (b) Key *i* moved to node *u* (c) Key *j* moved to parentOfu

FIGURE 42.19 The transfer operation transfers a key from the `parentOfu` to *u* and transfers a key from *u* 's sibling `parentOfu`.

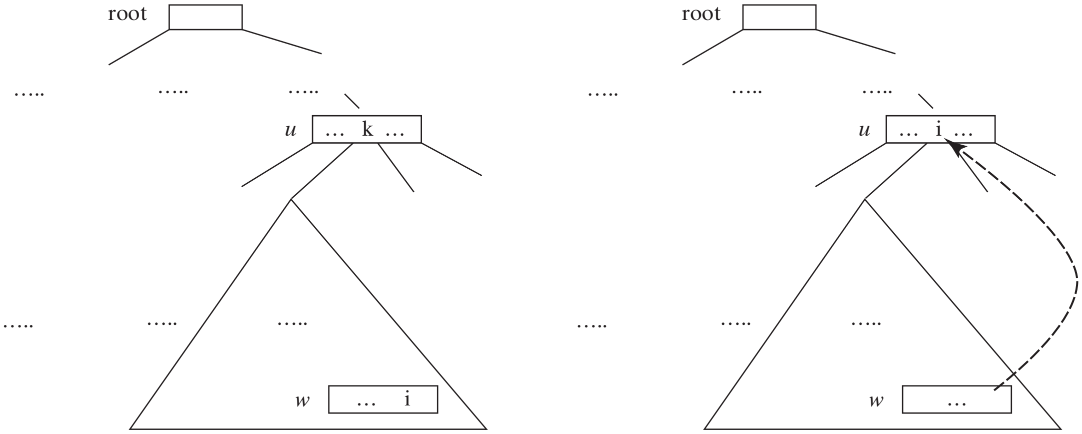
w of *u* that has more than $\lceil d/2 \rceil - 1$ keys if such sibling exists, as shown in Figure 42.19. Otherwise, perform a fusion with a sibling *w* of *u*, as shown in Figure 42.20.



(a) Before a fusion is performed (b) After a fusion is performed

FIGURE 42.20 The fusion operation moves key *i* from the `parentOfu` *u* to *w* and moves all keys in *u* to *w*.

Case 2: *u* is a nonleaf node. Find the rightmost leaf node in the left subtree of *k*. Let this node be *w*, as shown in Figure 42.21(a). Move the last key in *w* to replace *k* in *u*, as shown in Figure 42.21(b). If *w* becomes underflow, apply a transfer or fusion operation on *w*.



(a) Key is in *u* (b) Replace key *k* with key *i*

FIGURE 42.21 A key in the internal node is replaced by an element in a leaf node.

The performance of a B-tree depends on the number of disk IOs (i.e., the number of nodes accessed). The number of nodes accessed for search, insertion, and deletion operations depends on the height of the tree. In the worst case, each node contains $\lceil d/2 \rceil - 1$ keys. So, the height of the tree is $\log_{\lceil d/2 \rceil} n$, where *n* is the number of keys. In the best case, each node contains $d - 1$ keys. So, the height of the tree is $\log_d n$. Consider a B-tree of order 12 for 10 million keys. The height of the tree is between $\log_6 10{,}000{,}000 \approx 7$ and $\log_{12} 10{,}000{,}000 \approx 9$. So, for search, insertion, and deletion operations, the maximum number of nodes visited is 42. If you use an AVL tree, the maximum number of nodes visited is $\log_2 10{,}000{,}000 \approx 24$.

KEY TERMS

2–3–4 tree	42-2
2–4 tree	42-2
2-node	42-2
3-node	42-2
4-node	42-2
B-tree	42-11
fusion operation	42-7
split operation	42-4
transfer operation	42-7

CHAPTER SUMMARY

1. A 2–4 tree is a completely balanced search tree. In a 2–4 tree, a node may have one, two, or three elements.

2. Searching an element in a 2–4 tree is similar to searching an element in a binary tree. The difference is that you have searched an element within a node.

3. To insert an element to a 2–4 tree, locate a leaf node in which the element will be inserted. If the leaf node is a 2- or 3-node, simply insert the element into the node. If the node is a 4-node, split the node.

4. The process of deleting an element from a 2–4 tree is similar to that of deleting an element from a binary tree. The difference is that you have to perform transfer or fusion operations for empty nodes.

5. The height of a 2–4 tree is $O(\log n)$. So, the time complexities for the search, insert, and delete methods are $O(\log n)$.

6. A B-tree is a generalization of the 2–4 tree. Each node in a B-tree of order d can have between $\lceil d/2 \rceil - 1$ and $d - 1$ keys except the root. 2–4 trees are flatter than AVL trees and B-trees are flatter than 2–4 trees. B-trees are efficient for creating indexes for data in database systems where large amounts of data are stored on disks.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

***42.1** (*Implement `inorder`*) The `inorder` method in `Tree24` is left as an exercise. Implement it.

42.2 (*Implement `postorder`*) The `postorder` method in `Tree24` is left as an exercise. Implement it.

42.3 (*Implement `iterator`*) The `iterator` method in `Tree24` is left as an exercise. Implement it to iterate the elements using inorder.

***42.4** (*Display a 2–4 tree graphically*) Write a GUI program that displays a 2–4 tree.

*****42.5** (*2–4 tree animation*) Write a GUI program that animates the 2–4 tree `insert`, `delete`, and `search` methods, as shown in Figure 42.4.

****42.6** (*Parent reference for `Tree24`*) Redefine `Tree24Node` to add a reference to a node's parent, as shown below:

Tree24Node<E>	
elements: ArrayList<E>	An array list for storing the elements.
child: ArrayList<Tree24Node<E>>	An array list for storing the links to the child nodes.
parent: Tree24Node<E>	Refers to the parent of this node.
+Tree24()	Creates an empty tree node.
+Tree24(o: E)	Creates a tree node with an initial element.

Add the following two new methods in `Tree24`:

```
public Tree24Node<E> getParent(Tree24Node<E> node)
  Returns the parent for the specified node.
public ArrayList<Tree24Node<E>> getPath(Tree24Node<E> node)
  Returns the path from the specified node to the root in an
array list.
```

Write a test program that adds numbers `1`, `2`, ..., `100` to the tree and displays the paths for all leaf nodes.

*****42.7** (*The* `BTree` *class*) Design and implement a class for B-trees.

CHAPTER
43

RED-BLACK TREES

Objectives

- To know what a red-black tree is (§43.1).
- To convert a red-black tree to a 2–4 tree and vice versa (§43.2).
- To design the **`RBTree`** class that extends the **`BST`** class (§43.3).
- To insert an element in a red-black tree and resolve the double-red violation if necessary (§43.4).
- To delete an element from a red-black tree and resolve the double-black problem if necessary (§43.5).
- To implement and test the **`RBTree`** class (§§43.6–43.7).
- To compare the performance of AVL trees, 2–4 trees, and **`RBTree`** (§43.8).

43.1 Introduction

Key Point

A red-black tree is a balanced binary search tree derived from a 2–4 tree. A red-black tree corresponds to a 2-4 tree.

Each node in a red-black tree has a *color attribute* red or black, as shown in Figure 43.1(a). A node is called *external* if its left or right subtree is empty. Note that a leaf node is external, but an external node is not necessarily a leaf node. For example, node **25** is external, but it is not a leaf. The *black depth* of a node is defined as the number of black nodes in a path from the node to the root. For example, the black depth of node **25** is **2** and that of node **27** is **2**.

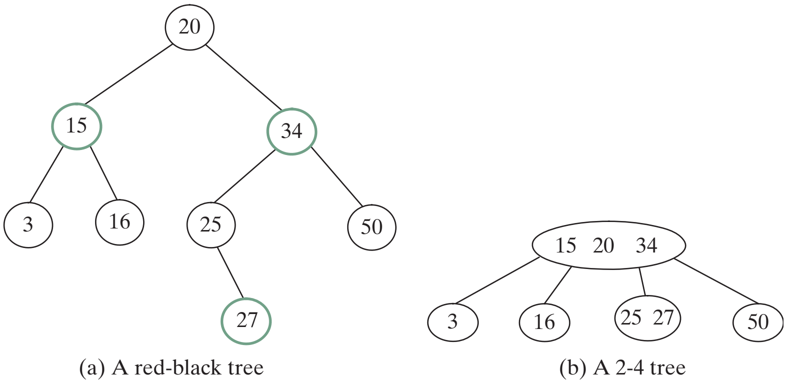
Note

The red nodes appear in blue in the text.

A red-black tree has the following properties:

1. The root is black.
2. Two adjacent nodes cannot be both red.
3. All external nodes have the same black depth.

The red-black tree in Figure 43.1(a) satisfies all three properties. A red-black tree can be converted to a 2-4 tree, and vice versa. Figure 43.1(b) shows an equivalent 2-4 tree for the red-black tree in Figure 43.1(a).



(a) A red-black tree

(b) A 2-4 tree

FIGURE 43.1 A red-black tree can be represented using a 2-4 tree, and vice versa.

43.2 Conversion between Red-Black Trees and 2-4 Trees

Key Point

This section discusses the correspondence between a red-black tree and a 2-4 tree.

You can design insertion and deletion algorithms for red-black trees without having knowledge of 2-4 trees. However, the correspondence between red-black trees and 2-4 trees provides useful intuition about the structure of red-black trees and operations. For this reason, this section discusses the correspondence between these two types of trees.

To convert a red-black tree to a 2-4 tree, simply merge every red node with its parent to create a 3-node or a 4-node. For example, the red nodes **15** and **34** are merged to their parent to create a 4-node, and the red node **27** is merged to its parent to create a 3-node, as shown in Figure 43.1(b).

To convert a 2-4 tree to a red-black tree, perform the following transformations for each node *u*:

1. If *u* is a 2-node, color it black, as shown in Figure 43.2(a).
2. If *u* is a 3-node with element values e_0 and e_1, there are two ways to convert it. Either make e_0 the parent of e_1 or make e_1 the parent of e_0. In any case, color the parent black and the child red, as shown in Figure 43.2(b).
3. If *u* is a 4-node with element values e_0, e_1, and e_2, make e_1 the parent of e_0 and e_2. Color e_1 black and e_0 and e_2 red, as shown in Figure 43.2(c).

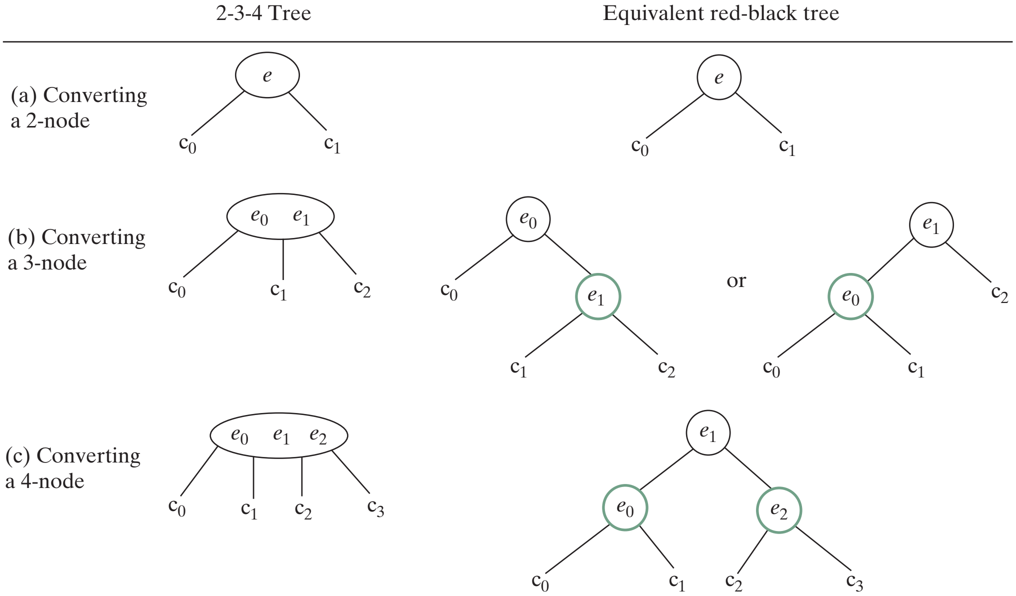

2-3-4 Tree
Equivalent red-black tree
(a) Converting a 2-node
(b) Converting a 3-node
(c) Converting a 4-node
e
c0
c1
e0 e1
c2
or
e0 e1 e2
c3

FIGURE 43.2 A node in a 2-4 tree can be transformed to nodes in a red-black tree.

Let us apply the transformation for the 2-4 tree in Figure 43.1(b). After transforming the 4-node, the tree is as shown in Figure 43.3(a). After transforming the 3-node, the tree is as shown in Figure 43.3(b). Note that the transformation for a 3-node is not unique. Therefore, the conversion from a 2-4 tree to a red-black tree is *not unique*. After transforming the 3-node, the tree could also be as shown in Figure 43.3(c).

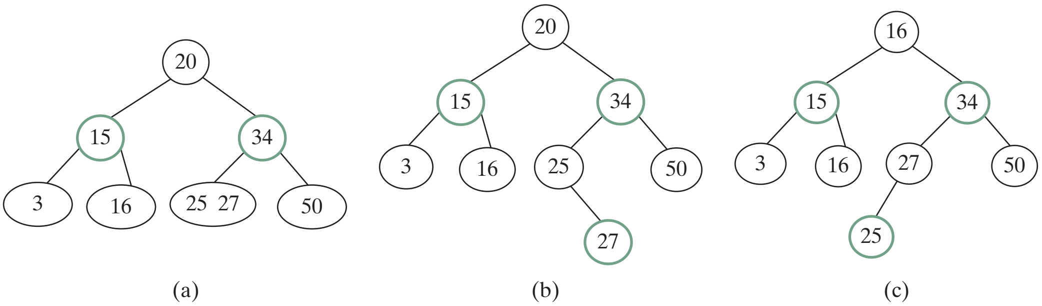

20
15
34
3
16
25 27
50
25
27
16
(a)
(b)
(c)

FIGURE 43.3 The conversion from a 2-4 tree to a red-black tree is not unique.

You can prove that the conversion results in a red-black tree that satisfies all three properties.

Property 1. The root is black.

Proof: If the root of a 2-4 tree is a 2-node, the root of the red-black tree is black. If the root of a 2-4 tree is a 3-node or 4-node, the transformation produces a black parent at the root.

Property 2. Two adjacent nodes cannot be both red.

Proof: Since the parent of a red node is always black, no two adjacent nodes can be both red.

Property 3. All external nodes have the same black depth.

Proof: When you covert a node in a 2-4 tree to red-black tree nodes, you get one black node and zero, one, or two red nodes as its children, depending on whether the original node is a 2-, 3-, or 4-node. Only a leaf 2-4 node may produce external red-black nodes. Since a 2-4 tree is perfectly balanced, the number of black nodes in any path from the root to an external node is the same.

Check Point

43.2.1 What is a red-black tree? What is an external node? What is black depth?

43.2.2 Describe the properties of a red-black tree.

43.2.3 How do you convert a red-black tree to a 2-4 tree? Is the conversion unique?

43.2.4 How do you convert a 2-4 tree to a red-black tree? Is the conversion unique?

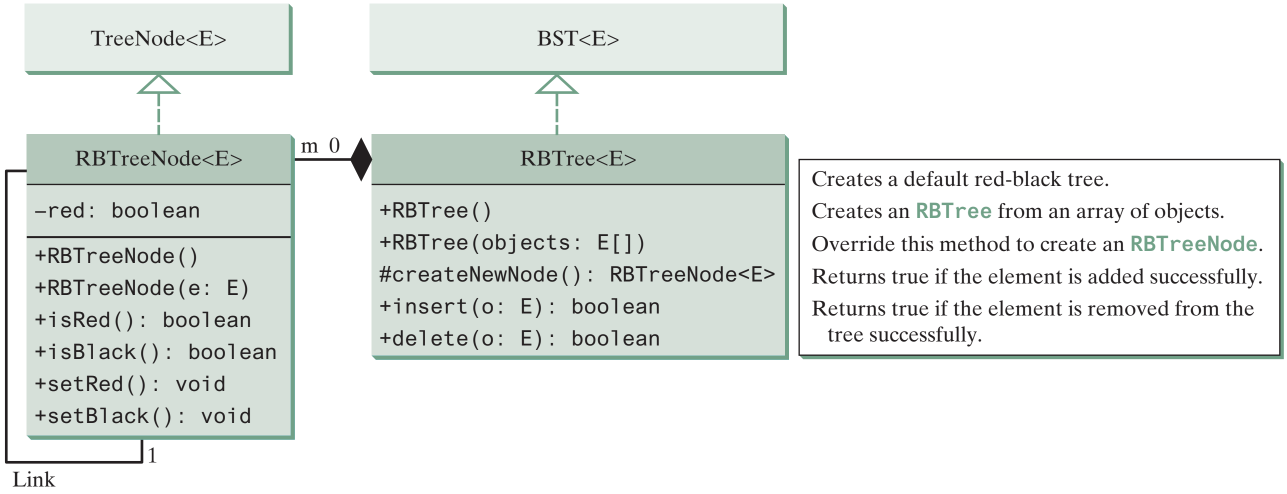
43.3 Designing Classes for Red-Black Trees

Key Point

A red-black tree designs a class for a red-black tree.

A red-black tree is a binary search tree. So, you can define the `RBTree` class to extend the `BST` class, as shown in Figure 43.4. The `BST` and `TreeNode` classes are defined in §26.2.5.

Each node in a red-black tree has a color property. Because the color is either red or black, it is efficient to use the `boolean` type to denote it. The `RBTreeNode` class can be defined to extend `BST.TreeNode` with the color property. For convenience, we also provide the methods for checking the color and setting a new color. Note that `TreeNode` is defined as a static inner class in `BST`. `RBTreeNode` will be defined as a static inner class in `RBTree`. Note that `BSTNode` contains the data fields `element`, `left`, and `right`, which are inherited in `RBTreeNode`. So, `RBTreeNode` contains four data fields, as pictured in Figure 43.5.



Creates a default red-black tree.
Creates an `RBTree` from an array of objects.
Override this method to create an `RBTreeNode`.
Returns true if the element is added successfully.
Returns true if the element is removed from the tree successfully.

FIGURE 43.4 The `RBTree` class extends `BST` with new implementations for the `insert` and `delete` methods.

`node: RBTreeNode<E>`

#element: E
-red: boolean
#left: TreeNode
#right: TreeNode

FIGURE 43.5 An `RBTreeNode` contains data fields `element`, `red`, `left`, and `right`.

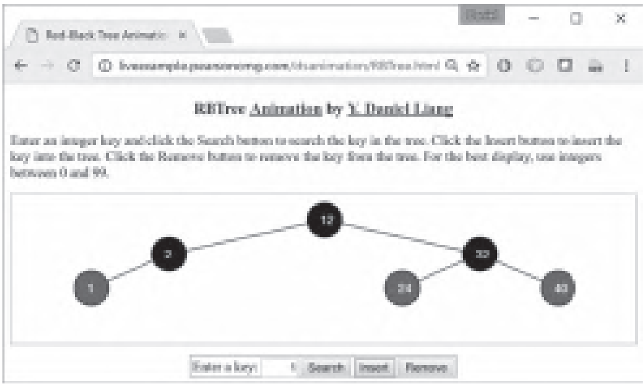
In the `BST` class, the `createNewNode()` method creates a `TreeNode` object. This method is overridden in the `RBTree` class to create an `RBTreeNode`. Note that the return type of the `createNewNode()` method in the `BST` class is `TreeNode`, but the return type of the `createNewNode()` method in `RBTree` class is `RBTreeNode`. This is fine, since `RBTreeNode` is a subtype of `TreeNode`.

Searching an element in a red-black tree is the same as searching in a regular binary search tree. So, the `search` method defined in the `BST` class also works for `RBTree`.

The `insert` and `delete` methods are overridden to insert and delete an element and perform operations for coloring and restructuring if necessary to ensure that the three properties of the red-black tree are satisfied.

Pedagogical Note

Run from http://liveexample.pearsoncmg.com/dsanimation/RBTree.html to see how a red-black tree works, as shown in Figure 43.6.



RBTree Animation by Y. Daniel Liang

Enter an integer key and click the Search button to search the key in the tree. Click the Insert button to insert the key into the tree. Click the Remove button to remove the key from the tree. For the best display, use integers between 0 and 99.

FIGURE 43.6 The animation tool enables you to insert, delete, and search elements in a red-black tree visually.

43.4 Overriding the `insert` Method

This section discusses how to insert an element to red-black tree.

Key Point

A new element is always inserted as a leaf node. If the new node is the root, color it black. Otherwise, color it red. If the parent of the new node is red, it violates Property 2 of the red-black tree. We call this a *double-red* violation.

Let u denote the new node inserted, v the parent of u, w the parent of v, and x the sibling of v. To fix the double-red violation, consider two cases:

Case 1: x is black or x is null. There are four possible configurations for u, v, w, and x, as shown in Figures 43.7(a), 43.8(a), 43.9(a), and 43.10(a). In this case, u, v, and w form a 4-node in the corresponding 2-4 tree, as shown in Figures 43.7(c), 43.8(c), 43.9(c), and 43.10(c), but are represented incorrectly in the red-black tree. To correct this error, restructure and recolor three nodes u, v, and w, as shown in Figures 43.7(b), 43.8(b), 43.9(b), and 43.10(b). Note that x, y_1, y_2, and y_3 may be null.

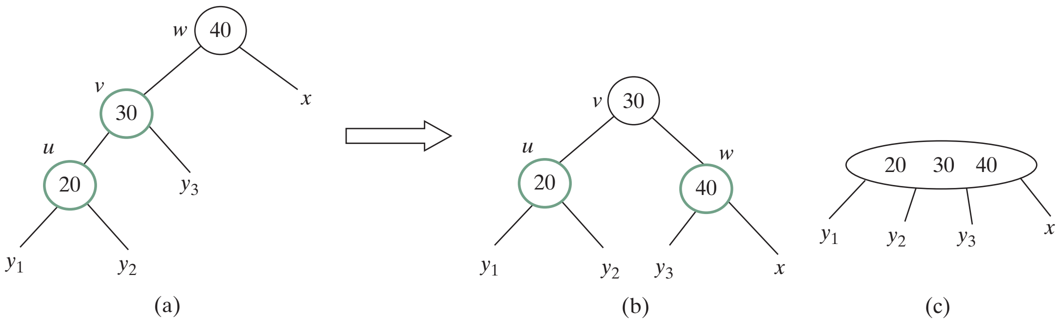

w 40
v 30
u 20
x
y_3
y_1
y_2
(a)
v 30
u 20
w 40
y_1
y_2
y_3
x
(b)
20 30 40
y_1
y_2
y_3
x
(c)

FIGURE 43.7 Case 1.1: $u < v < w$.

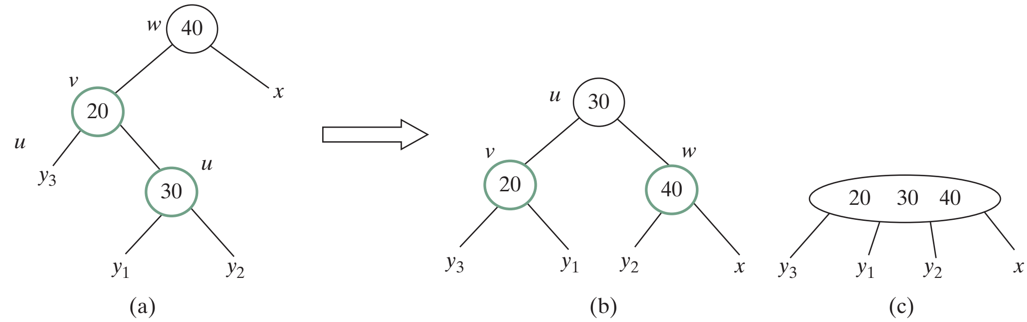

w 40
v 20
u
x
y_3
u 30
y_1
y_2
(a)
u 30
v 20
w 40
y_3
y_1
y_2
x
(b)
20 30 40
y_3
y_1
y_2
x
(c)

FIGURE 43.8 Case 1.2: $v < u < w$

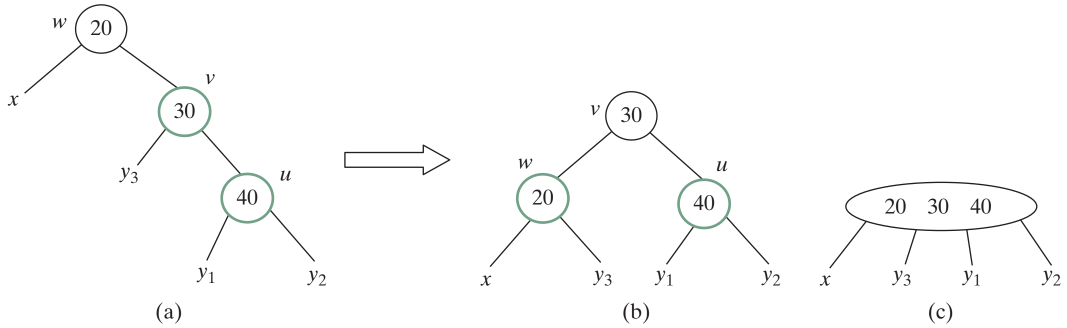

w 20
x
v 30
y_3
u 40
y_1
y_2
(a)
v 30
w 20
u 40
x
y_3
y_1
y_2
(b)
20 30 40
x
y_3
y_1
y_2
(c)

FIGURE 43.9 Case 1.3: $w < v < u$

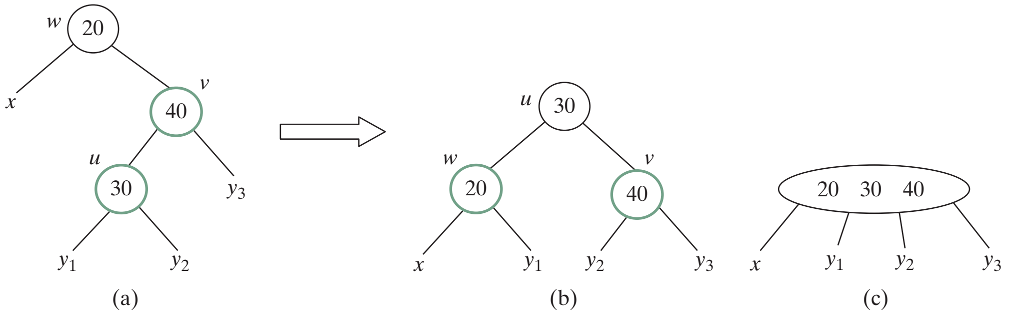
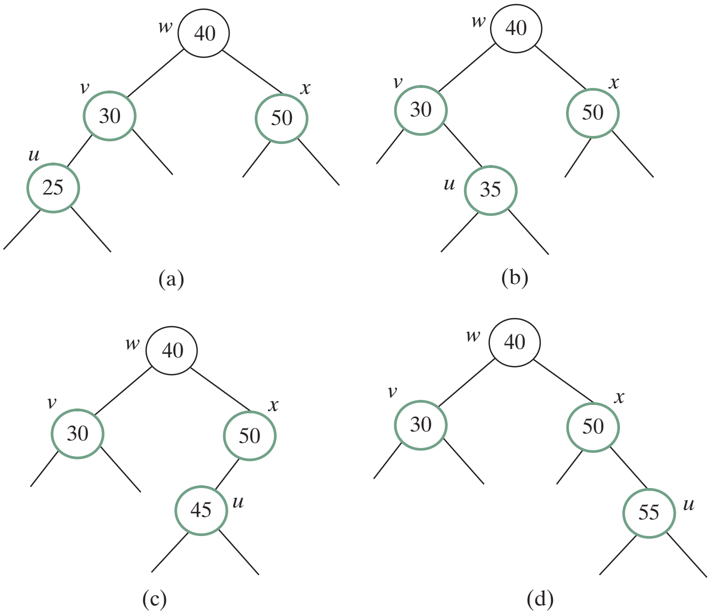

w 20
x
v 40
u 30
y_3
y_1
y_2
(a)
u 30
w 20
v 40
x
y_1
y_2
y_3
(b)
20 30 40
x
y_1
y_2
y_3
(c)

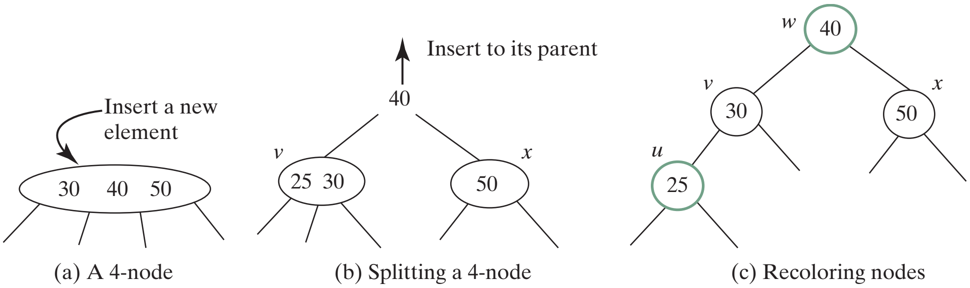
FIGURE 43.10 Case 1.4: $w < u < v$

Case 2: x is red. There are four possible configurations for u, v, w, w, and x, as shown in Figures 43.11(a), 43.11(b), 43.11(c), and 43.11(d). All of these configurations correspond to an overflow situation in the corresponding 4-node in a 2-4 tree, as shown in Figure 43.12(a). A splitting operation is performed to fix the overflow problem in a 2-4 tree, as shown in Figure 43.12(b). We perform an equivalent recoloring operation to fix the problem in a red-black tree. Color w and u red and color two children of w black. Assume u is a left child of v, as shown



w
40
v
30
x
50
u
25
(a)
w
40
v
30
x
50
u
35
(b)
w
40
v
30
x
50
45
u
(c)
w
40
v
30
x
50
55
u
(d)

FIGURE 43.11 Case 2 has four possible configurations.



Insert a new element
30 40 50
(a) A 4-node
Insert to its parent
40
v
25 30
x
50
(b) Splitting a 4-node
w
40
v
30
x
50
u
25
(c) Recoloring nodes

FIGURE 43.12 Splitting a 4-node corresponds to recoloring the nodes in the red-black tree.

in Figure 43.11(a). After recoloring, the nodes are shown in Figure 43.12(c). Now *w* is red, if *w*'s parent is black, the double-red violation is fixed. Otherwise, a new double-red violation occurs at node *w*. We need to continue the same process to eliminate the double-red violation at *w*, recursively.

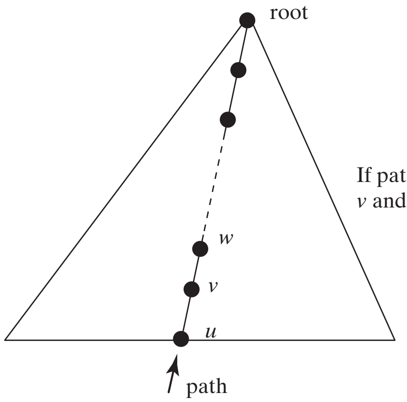
A more detailed algorithm for inserting an element is described in Listing 43.1.

LISTING 43.1 Inserting an Element to a Red-Black Tree

```
public boolean insert(E e) {
  boolean successful = super.insert(e);
  if (!successful)
    return false; // e is already in the tree
  else {
    ensureRBTree(e);
  }

  return true; // e is inserted
}

/** Ensure that the tree is a red-black tree */
private void ensureRBTree(E e) {
  Get the path that leads to element e from the root.
15      int i = path.size() – 1; // Index to the current node in the path
16      Get u, v from the path. u is the node that contains e and v
17        is the parent of u.
18      Color u red;
19
20      if (u == root) // If e is inserted as the root, set root black
21        u.setBlack();
22      else if (v.isRed())
23        fixDoubleRed(u, v, path, i); // Fix double-red violation at u
24    }
25
26    /** Fix double-red violation at node u */
27    private void fixDoubleRed(RBTreeNode<E> u, RBTreeNode<E> v,
28        ArrayList<TreeNode<E>> path, int i) {
29      Get w from the path. w is the grandparent of u.
30
31      // Get v’s sibling named x
32      RBTreeNode<E> x = (w.left == v) ?
33        (RBTreeNode<E>)(w.right) : (RBTreeNode<E>)(w.left);
34
35      if (x == null || x.isBlack()) {
36        // Case 1: v's sibling x is black
37        if (w.left == v && v.left == u) {
38          // Case 1.1: u < v < w, Restructure and recolor nodes
39        }
40        else if (w.left == v && v.right == u) {
41          // Case 1.2: v < u < w, Restructure and recolor nodes
42        }
43        else if (w.right == v && v.right == u) {
44          // Case 1.3: w < v < u, Restructure and recolor nodes
45        }
46        else {
47          // Case 1.4: w < u < v, Restructure and recolor nodes
48        }
49      }
50      else { // Case 2: v's sibling x is red
51        Color w and u red
52        Color two children of w black.
53
54        if (w is root) {
55          Set w black;
56        }
57        else if (the parent of w is red) {
58          // Propagate along the path to fix new double-red violation
59          u = w;
60          v = parent of w;
61          fixDoubleRed(u, v, path, i – 2); // i – 2 propagates upward
62        }
63      }
64    }
```

The `insert(E e)` method (lines 1–10) invokes the `insert` method in the `BST` class to create a new leaf node for the element (line 2). If the element is already in the tree, return false (line 4). Otherwise, invoke `ensureRBTree(e)` (line 6) to ensure that the tree satisfies the color and black depth property of the red-black tree.

The `ensureRBTree(E e)` method (lines 13–24) obtains the path that leads to `e` from the root (line 14), as shown in Figure 43.13. This path plays an important role to implement the algorithm. From this path, you get nodes `u` and `v` (lines 16–17). If `u` is the root, color `u` black (lines 20–21). If `v` is red, a double-red violation occurs at node `u`. Invoke `fixDoubleRed` to fix the problem.



If path.get(i) is *u*, path.get(i – 1) is *v* and path.get(i – 2) is *w*.

FIGURE 43.13 The path consists of the nodes from *u* to the root.

The `fixDoubleRed` method (lines 27–63) fixes the double-red violation. It first obtains `w` (the parent of `v`) from the path (line 29) and `x` (the sibling of `v`) (lines 32–33). If `x` is empty or a black node, restructure and recolor three nodes `u`, `v`, and `w` to eliminate the problem (lines 35–49). If `x` is a red node, recolor the nodes `u`, `v`, `w,` and `x` (lines 51–52). If `w` is the root, color `w` black (lines 54–56). If the parent of `w` is red, the double-red violation reappears at `w`. Invoke `fixDoubleRed` with new `u` and `v` to fix the problem (line 61). Note that now `i – 2` points to the new `u` in the path. This adjustment is necessary to locate the new nodes `w` and parent of `w` along the path.

Figure 43.14 shows the steps of inserting `34`, `3`, `50`, `20`, `15`, `16`, `25`, and `27` into an empty red-black tree. When inserting `20` into the tree in (d), Case 2 applies to recolor `3` and `50` to black. When inserting `15` into the tree in (g), Case 1.4 applies to restructure and recolor nodes `15`, `20`, and `3`. When inserting `16` into the tree in (i), Case 2 applies to recolor nodes `3` and `20` to black and nodes `15` and `16` to red. When inserting `27` into the tree in (l), Case 2 applies to recolor nodes `16` and `25` to black and nodes `20` and `27` to red. Now a new double-red problem occurs at node `20`. Apply Case 1.2 to restructure and recolor nodes. The new tree is shown in (n).

43.5 Overriding the `delete` Method

This section discusses how to delete an element to red-black tree.

Key Point

To delete an element from a red-black tree, first search the element in the tree to locate the node that contains the element. If the element is not in the tree, the method returns false. Let *u* be the node that contains the element. If *u* is an internal node with both left and right children, find the rightmost node in the left subtree of *u*. Replace the element in *u* with the element in the rightmost node. Now we will only consider deleting external nodes.

Let *u* be an external node to be deleted. Since *u* is an external node, it has at most one child, denoted by *childOfu*. *childOfu* may be `null`. Let *parentOfu* denote the parent of *u*, as shown in Figure 43.15(a). Delete *u* by connecting *childOfu* with *parentOfu*, as shown in Figure 43.15(b).

Consider the following case:

- If *u* is red, we are done.
- If *u* is black and *childOfu* is red, color *childOfu* black to maintain the black height for *childOfu*.
- Otherwise, assign *childOfu* a fictitious *double black*, as shown in Figure 43.16(a). We call this a *double-black problem*, which indicates that the black depth is short by `1`, caused by deleting a black node `u`.

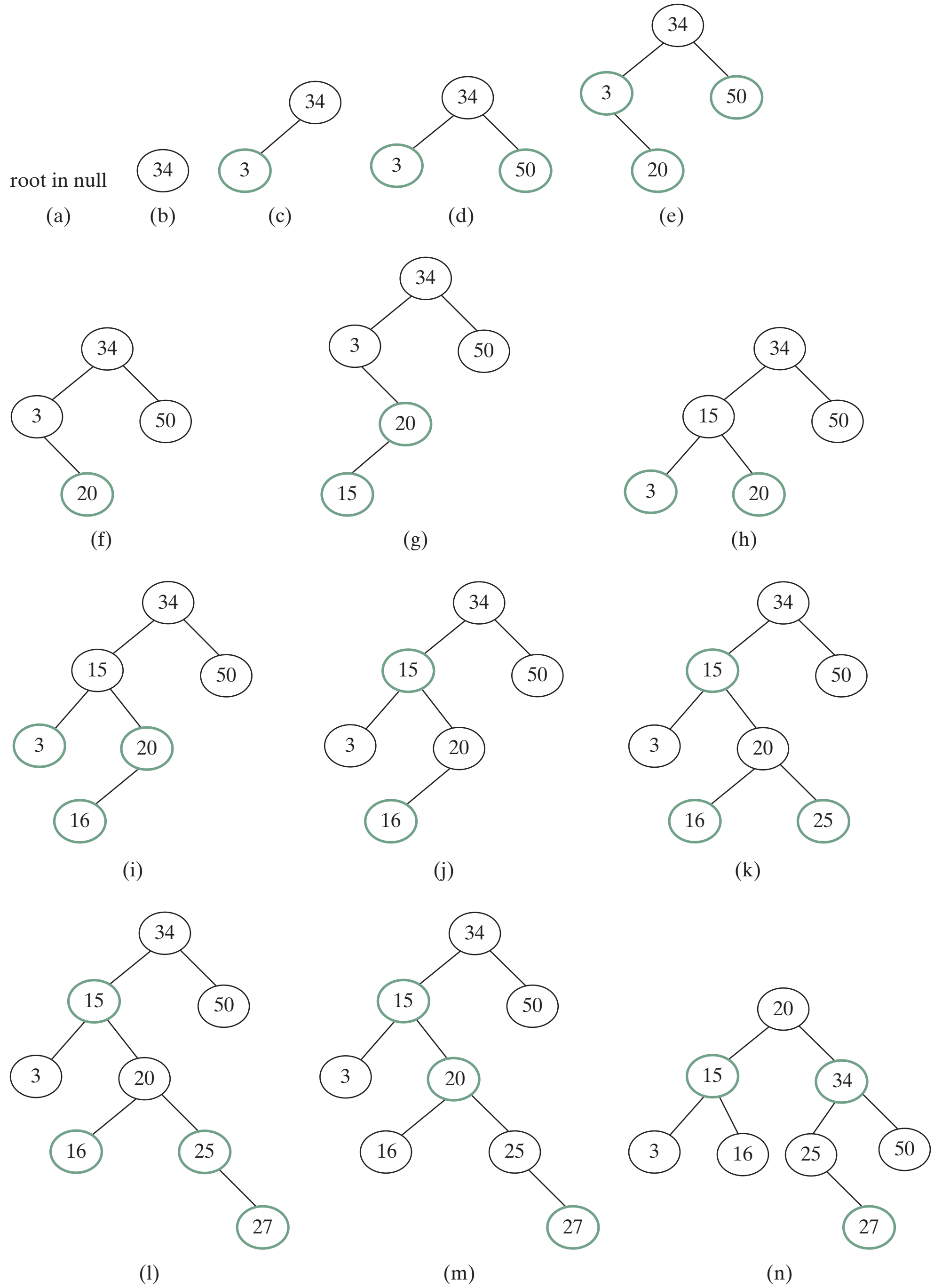
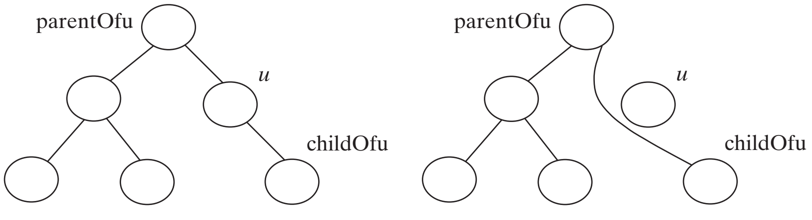
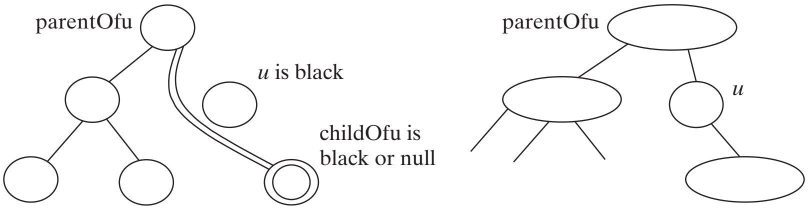

root in null
(a)
34
(b)
34
3
(c)
34
3
50
(d)
34
3
50
20
(e)
34
3
50
20
(f)
34
3
50
20
15
(g)
34
15
50
3
20
(h)
34
15
50
3
20
16
(i)
34
15
50
3
20
16
(j)
34
15
50
3
20
16
25
(k)
34
15
50
3
20
16
25
27
(l)
34
15
50
3
20
16
25
27
(m)
20
15
34
3
16
25
50
27
(n)

Figure 43.14 Inserting into a red-black tree: (a) initial empty tree; (b) inserting **34**; (c) inserting **3**; (d) inserting **50**; (e) inserting **20** causes a double red; (f) after recoloring (Case 2); (g) inserting **15** causes a double red; (h) after restructuring and recoloring (Case 1.4); (i) inserting **16** causes a double red; (j) after recoloring (Case 2); (k) inserting **25**; (l) inserting **27** causes a double red at **27**; (m) a double red at 20 reappears after recoloring (Case 2); and (n) after restructuring and recoloring (Case 1.2).


parentOfu
u
childOfu
parentOfu
u
childOfu

(a) Before deleting *u* (b) After deleting *u*

FIGURE 43.15 `u` is an external node and `childOfu` may be null.

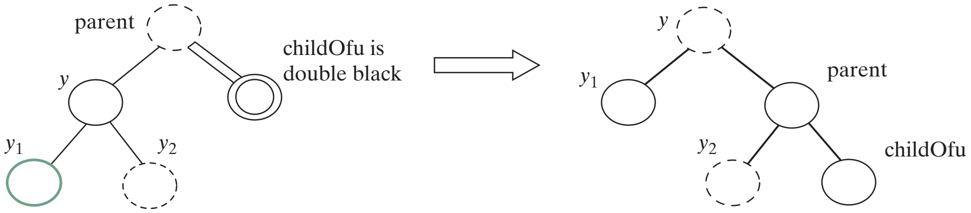

parentOfu
u is black
childOfu is
black or null
parentOfu
u

(a) (b)

FIGURE 43.16 (a) `childOfu` is denoted double black. (b) *u* corresponds to an empty node in a 2-4 tree.

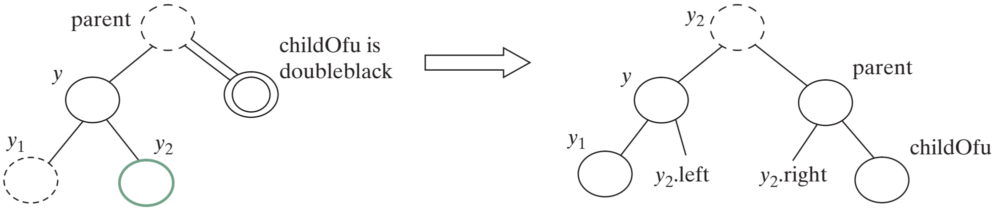
A double black in a red-black tree corresponds to an empty node for `u` (i.e., underflow situation) in the corresponding 2-4 tree, as shown in Figure 43.16(b). To fix the double-black problem, we will perform equivalent transfer and fusion operations. Consider three cases:

Case 1: The sibling `y` of *childOfu* is black and has a red child. This case has four possible configurations, as shown in Figures 43.17(a), 43.18(a), 43.19(a), and 43.20(a). The dashed circle denotes that the node is either red or black. To eliminate the double-black problem, restructure and recolor the nodes, as shown in Figures 43.17(b), 43.18(b), 43.19(b), and 43.20(b).


parent
childOfu is
double black
y
y_1
y_2
y
y_1
parent
y_2
childOfu

(a) (b)

FIGURE 43.17 Case 1.1: The sibling `y` of `childOfu` is black and `y1` is red.


parent
childOfu is
doubleblack
y
y_1
y_2
y_2
y
parent
y_1
childOfu
y_2.left
y_2.right

(a) (b)

FIGURE 43.18 Case 1.2: The sibling `y` of `childOfu` is black and `y2` is red.

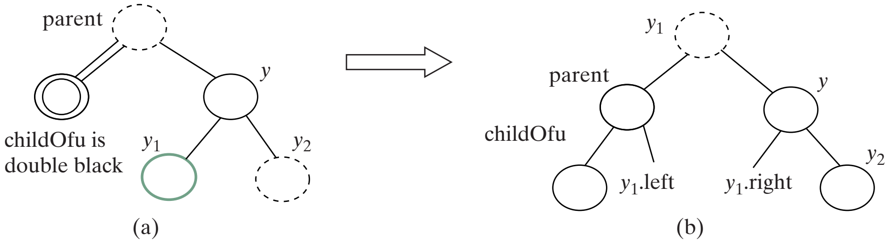

parent
y
childOfu is
double black
y_1
y_2
(a)
y_1
parent
y
childOfu
y_1.left
y_1.right
y_2
(b)

FIGURE 43.19 Case 1.3: The sibling `y` of `childOfu` is black and `y1` is red.

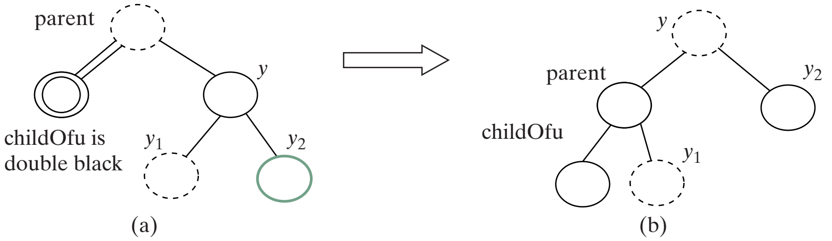

parent
y
childOfu is
double black
y_1
y_2
(a)
y
parent
y_2
childOfu
y_1
(b)

FIGURE 43.20 Case 1.4: the sibling `y` of `childOfu` is black and `y2` is red.

Note

Case 1 corresponds to a *transfer* operation in the 2-4 tree. For example, the corresponding 2-4 tree for Figure 43.17(a) is shown in Figure 43.21(a), and it is transformed into Figure 43.21(b) through a transfer operation.

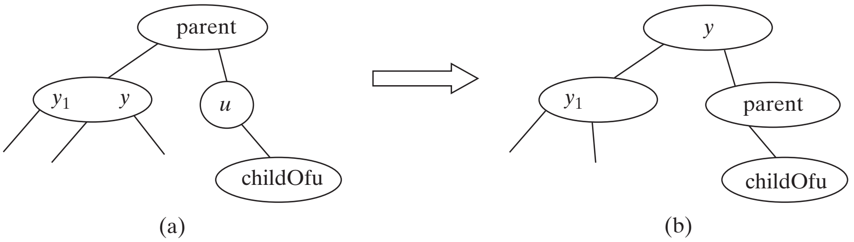

parent
y_1
y
u
childOfu
(a)
y
y_1
parent
childOfu
(b)

FIGURE 43.21 Case 1 corresponds to a transfer operation in the corresponding 2-4 tree.

Case 2: The sibling `y` of *childOfu* is black and its children are black or `null`. In this case, change `y`'s color to red. If `parent` is red, change it to black, and we are done, as shown in Figure 43.22. If `parent` is black, we denote `parent` double black, as shown in Figure 43.23. The double-black problem *propagates* to the parent node.

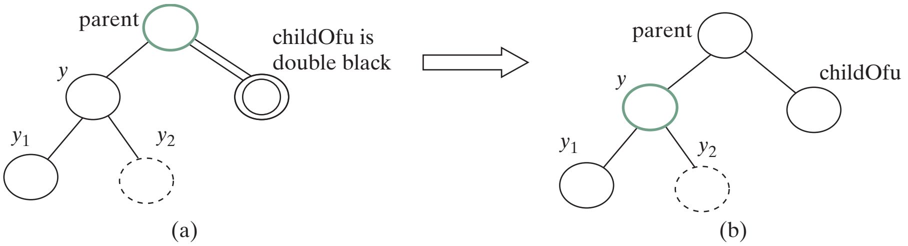

parent
childOfu is
double black
y
y_1
y_2
(a)
parent
childOfu
y
y_1
y_2
(b)

FIGURE 43.22 Case 2: Recoloring eliminates the double-black problem if `parent` is red.

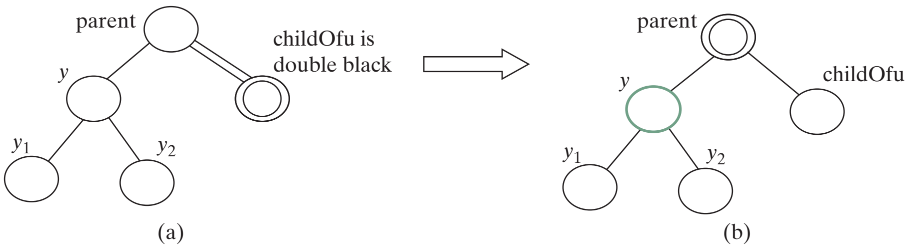

parent
childOfu is
double black
y
y_1
y_2
(a)
parent
childOfu
y
y_1
y_2
(b)

Figure 43.23 Case 2: Recoloring propagates the double-black problem if `parent` is black.

Note
Figures 43.22 and 43.22 show that `childOfu` is a right child of `parent`. If `childOfu` is a left child of `parent`, recoloring is performed identically.

Note
Case 2 corresponds to a *fusion* operation in the 2-4 tree. For example, the corresponding 2-4 tree for Figure 43.22(a) is shown in Figure 43.24(a), and it is transformed into Figure 43.24(b) through a fusion operation.

Case 3: The sibling `y` of *childOfu* is red. In this case, perform an *adjustment* operation. If `y` is a left child of `parent`, let `y1` and `y2` be the left and right children of `y`, as shown in Figure 43.25. If `y` is a right children of `parent`, let `y1` and `y2` be the left and right child of `y`, as shown in Figure 43.26. In both cases, color `y` black and `parent` red. `childOfu` is still a fictitious double-black node. After the adjustment, the sibling of `childOfu` is now black, and either Case 1 or Case 2 applies. If Case 1 applies, a one-time restructuring and recoloring operation eliminates the double-black problem. If Case 2 applies, the double-black problem cannot reappear, since `parent` is now red. Therefore, one-time application of Case 1 or Case 2 will complete Case 3.

Note
Case 3 results from the fact that a 3-node may be transformed in two ways to a red-black tree, as shown in Figure 43.27.

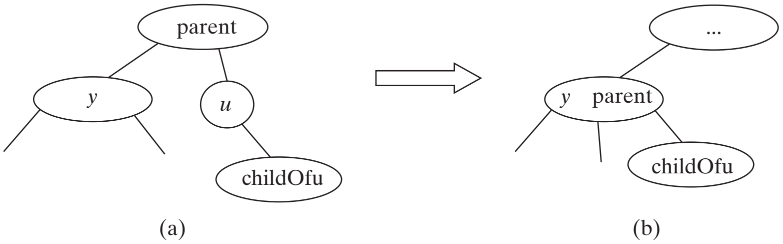

parent
y
u
childOfu
(a)
...
y parent
childOfu
(b)

Figure 43.24 Case 2 corresponds to a fusion operation in the corresponding 2-4 tree.

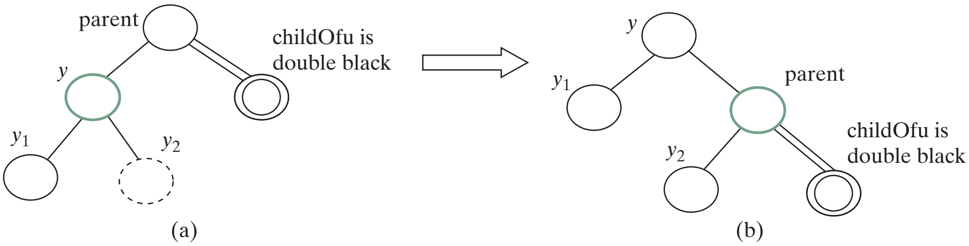

parent
childOfu is
double black
y
y_1
y_2
(a)
y
y_1
parent
childOfu is
double black
y_2
(b)

Figure 43.25 Case 3.1: `y` is a left red child of parent.

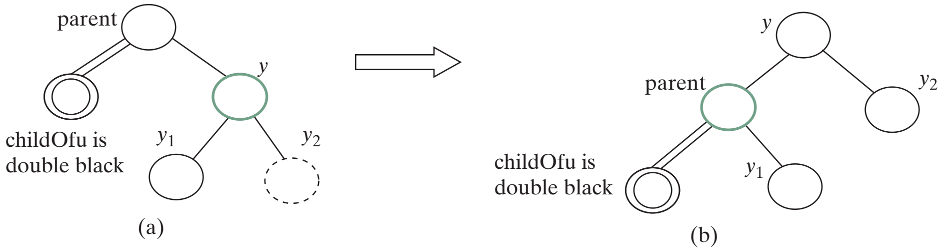

parent
y
childOfu is
double black
y_1
y_2
(a)
y
parent
y_2
childOfu is
double black
y_1
(b)

FIGURE 43.26 Case 3.2: `y` is a right red child of `parent`.

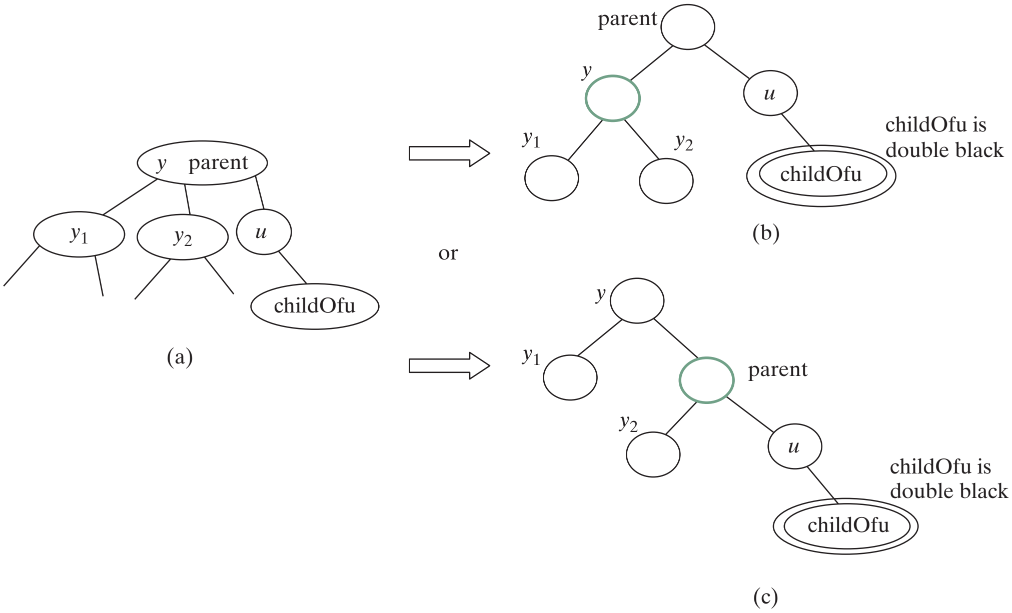

y parent
y_1
y_2
u
childOfu
(a)
or
parent
y
u
y_1
y_2
childOfu
childOfu is
double black
(b)
y
y_1
parent
y_2
u
childOfu is
double black
childOfu
(c)

FIGURE 43.27 A 3-node may be transformed in two ways to red-black tree nodes.

Based on the foregoing discussion, Listing 43.2 presents a more detailed algorithm for deleting an element.

LISTING 43.2 Deleting an Element from a Red-Black Tree

```
public boolean delete(E e) {
  Locate the node to be deleted
  if (the node is not found)
    return false;

  if (the node is an internal node) {
    Find the rightmost node in the subtree of the node;
    Replace the element in the node with the one in rightmost;
    The rightmost node is the node to be deleted now;
  }

  Obtain the path from the root to the node to be deleted;

  // Delete the last node in the path and propagate if needed
  deleteLastNodeInPath(path);

  size--; // After one element deleted
  return true; // Element deleted
}

/** Delete the last node from the path. */
public void deleteLastNodeInPath(ArrayList<TreeNode<e>> path) {
  Get the last node u in the path;
  Get parentOfu and grandparentOfu in the path;
  Get childOfu from u;
  Delete node u. Connect childOfu with parentOfu

  // Recolor the nodes and fix double black if needed
  if (childOfu == root || u.isRed())
    return; // Done if childOfu is root or if u is red
  else if (childOfu != null && childOfu.isRed())
    childOfu.setBlack(); // Set it black, done
  else // u is black, childOfu is null or black
    // Fix double black on parentOfu
    fixDoubleBlack(grandparentOfu, parentOfu, childOfu, path, i);
  }

  /** Fix the double black problem at node parent */
  private void fixDoubleBlack(
      RBTreeNode<E> grandparent, RBTreeNode<E> parent,
      RBTreeNode<E> db, ArrayList<TreeNode<E>> path, int i) {
    Obtain y, y1, and y2

    if (y.isBlack() && y1 != null && y1.isRed()) {
      if (parent.right == db) {
        // Case 1.1: y is a left black sibling and y1 is red
        Restructure and recolor parent, y, and y1 to fix the problem;
      }
      else {
        // Case 1.3: y is a right black sibling and y1 is red
        Restructure and recolor parent, y1, and y to fix the problem;
      }
    }
    else if (y.isBlack() && y2 != null && y2.isRed()) {
      if (parent.right == db) {
        // Case 1.2: y is a left black sibling and y2 is red
        Restructure and recolor parent, y2, and y to fix the problem;
      }
      else {
        // Case 1.4: y is a right black sibling and y2 is red
        Restructure and recolor parent, y, and y2 to fix the problem;
      }
    }
    else if (y.isBlack()) {
      // Case 2: y is black and y's children are black or null
      Recolor y to red;

      if (parent.isRed())
        parent.setBlack(); // Done
      else if (parent != root) {
        // Propagate double black to the parent node
        // Fix new appearance of double black recursively
        db = parent;
        parent = grandparent;
        grandparent =
          (i >= 3) ? (RBTreeNode<E>)(path.get(i - 3)) : null;
77        fixDoubleBlack(grandparent, parent, db, path, i - 1);
78      }
79    }
80    else if (y.isRed()) {
81      if (parent.right == db) {
82        // Case 3.1: y is a left red child of parent
83        parent.left = y2;
84        y.right = parent;
85      }
86      else {
87        // Case 3.2: y is a right red child of parent
88        parent.right = y.left;
89        y.left = parent;
90      }
91
92      parent.setRed(); // Color parent red
93      y.setBlack(); // Color y black
94      connectNewParent(grandparent, parent, y); // y is new parent
95      fixDoubleBlack(y, parent, db, path, i - 1);
96    }
97  }
```

The `delete(E e)` method (lines 1–19) locates the node that contains `e` (line 2). If the node does not exist, return `false` (lines 3–4). If the node is an internal node, find the right most node in its left subtree and replace the element in the node with the element in the right most node (lines 6–9). Now the node to be deleted is an external node. Obtain the path from the root to the node (line 12). Invoke `deleteLastNodeInPath(path)` to delete the last node in the path and ensure that the tree is still a red-black tree (line 15).

The `deleteLastNodeInPath` method (lines 22–36) obtains the last node `u`, `parentOfu`, `grandparendOfu`, and `childOfu` (lines 23–26). If `childOfu` is the root or `u` is red, the tree is fine (lines 29–30). If `childOfu` is red, color it black (lines 31–32). We are done. Otherwise, `u` is black and `childOfu` is `null` or black. Invoke `fixDoubleBlack` to eliminate the double-black problem (line 35).

The `fixDoubleBlack` method (lines 39–97) eliminates the double-black problem. Obtain `y`, `y1`, and `y2` (line 42). `y` is the sibling of the double-black node. `y1` and `y2` are the left and right children of `y`. Consider three cases:

1. If `y` is black and one of its children is red, the double-black problem can be fixed by one-time restructuring and recoloring in Case 1 (lines 44–63).

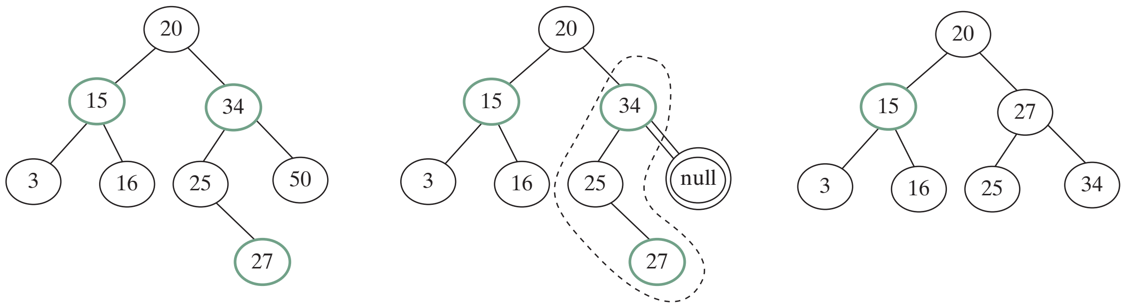
2. If `y` is black and its children are `null` or black, change `y` to red. If `parent` of `y` is black, denote `parent` to be the new double-black node and invoke `fixDoubleBlack` recursively (line 77).

3. If `y` is red, adjust the nodes to make `parent` a child of `y` (lines 84, 89) and color `parent` red and `y` black (lines 92–93). Make `y` the new parent (line 94). Recursively invoke `fixDoubleBlack` on the same double-black node with a different color for `parent` (line 95).

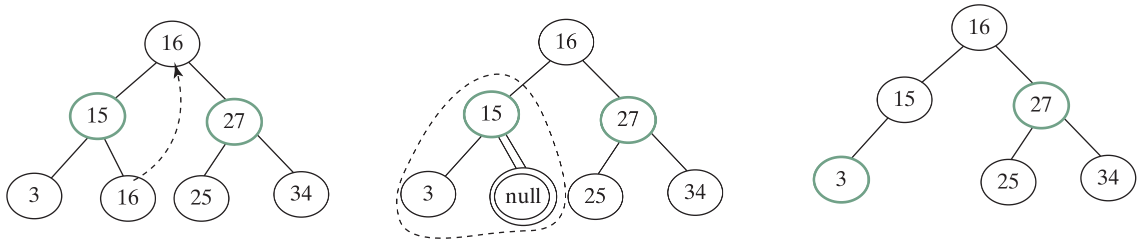
Figure 43.28 shows the steps of deleting elements. To delete `50` from the tree in Figure 43.28(a), apply Case 1.2, as shown in Figure 43.28(b). After restructuring and recoloring, the new tree is as shown in Figure 43.28(c).

When deleting `20` in Figure 43.28(c), 20 is an internal node, and it is replaced by `16`, as shown in Figure 43.28(d). Now Case 2 applies to deleting the rightmost node, as shown in Figure 43.28(e). Recolor the nodes results in a new tree, as shown in Figure 43.28(f).

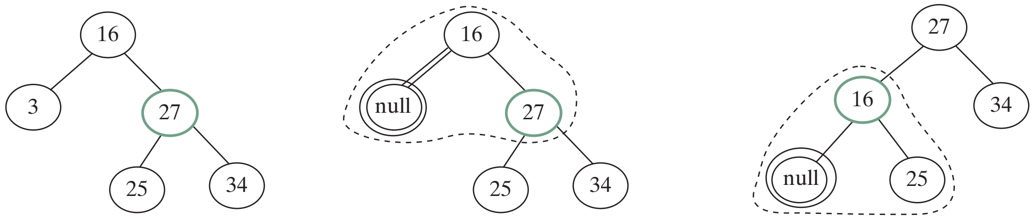
When deleting `15`, connect node 3 with node 20 and color node 3 black, as shown in Figure 43.28(g). We are done.


20
15
34
3
16
25
50
27
20
15
34
3
16
25
null
27
20
15
27
3
16
25
34

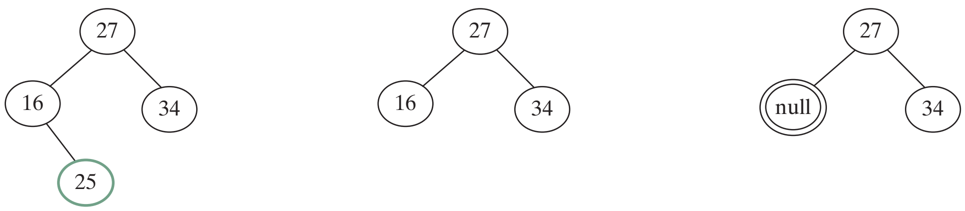
(a) Delete 50 (b) Case 1.2 (c) Delete 20


16
15
27
3
16
25
34
16
15
27
3
null
25
34
16
15
27
3
25
34

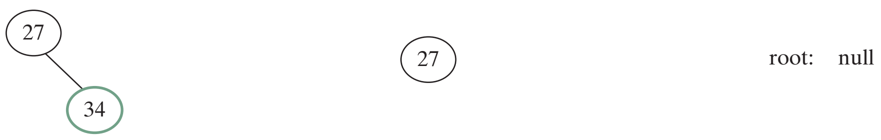
(d) Copy 16 to replace 20 (e) Case 2 (f) Delete 15


16
3
27
25
34
16
null
27
25
34
27
16
34
null
25

(g) Delete 3 (h) Case 3 (i) Case 2


27
16
34
25
27
16
34
27
null
34

(j) Delete 25 (k) Delete 16 (l) Case 2


27
34
27
root: null

(m) Delete 34 (n) Delete 27 (o) Empty tree

FIGURE 43.28 Delete elements from a red-black tree.

After deleting **25**, the new tree is as shown in Figure 43.28(j). Now delete **16**. Apply Case 2, as shown in Figure 43.28(k). The new tree is shown in Figure 43.28(l).

After deleting **34**, the new tree is as shown in Figure 43.28(m).

After deleting **27**, the new tree is as shown in Figure 43.28(n).

Check Point

43.5.1 What are the data fields in `RBTreeNode`?

43.5.2 How do you insert an element into a red-black tree and how do you fix the double-red violation?

43.5.3 How do you delete an element from a red-black tree and how do you fix the double-black problem?

43.5.4 Show the change of the tree when inserting `1`, `2`, `3`, `4`, `10`, `9`, `7`, `5`, `8`, and `6` into it, in this order.

43.5.5 For the tree built in the preceding question, show the change of the tree after deleting `1`, `2`, `3`, `4`, `10`, `9`, `7`, `5`, `8`, and `6` from it in this order.

43.6 Implementing `RBTree` Class

Key Point

This section implements the `RBTree` class.

Listing 43.3 gives a complete implementation for the `RBTree` class.

LISTING 43.3 RBTree.java

```
import java.util.ArrayList;

public class RBTree<E extends Comparable<E>> extends BST<E> {
  /** Create a default RB tree */
  public RBTree() {
  }

  /** Create an RB tree from an array of elements */
  public RBTree(E[] elements) {
    super(elements);
  }

  @Override /** Override createNewNode to create an RBTreeNode */
  protected RBTreeNode<E> createNewNode(E e) {
    return new RBTreeNode<E>(e);
  }

  @Override /** Override the insert method to
    balance the tree if necessary */
  public boolean insert(E e) {
    boolean successful = super.insert(e);
    if (!successful)
      return false; // e is already in the tree
    else {
      ensureRBTree(e);
    }

    return true; // e is inserted
  }

  /** Ensure that the tree is a red-black tree */
  private void ensureRBTree(E e) {
    // Get the path that leads to element e from the root
    ArrayList<TreeNode<E>> path = path(e);

    int i = path.size() - 1; // Index to the current node in the path

    // u is the last node in the path. u contains element e
    RBTreeNode<E> u = (RBTreeNode<E>)(path.get(i));

    // v is the parent of of u, if exists
    RBTreeNode<E> v = (u == root) ? null :
      (RBTreeNode<E>)(path.get(i - 1));

    u.setRed(); // It is OK to set u red

    if (u == root) // If e is inserted as the root, set root black
      u.setBlack();
    else if (v.isRed())
      fixDoubleRed(u, v, path, i); // Fix double-red violation at u
  }

  /** Fix double-red violation at node u */
  private void fixDoubleRed(RBTreeNode<E> u, RBTreeNode<E> v,
      ArrayList<TreeNode<E>> path, int i) {
    // w is the grandparent of u
    RBTreeNode<E> w = (RBTreeNode<E>)(path.get(i - 2));
    RBTreeNode<E> parentOfw = (w == root) ? null :
      (RBTreeNode<E>)path.get(i - 3);

    // Get v's sibling named x
    RBTreeNode<E> x = (w.left == v) ?
      (RBTreeNode<E>)(w.right) : (RBTreeNode<E>)(w.left);

    if (x == null || x.isBlack()) {
      // Case 1: v's sibling x is black
      if (w.left == v && v.left == u) {
        // Case 1.1: u < v < w, Restructure and recolor nodes
        restructureRecolor(u, v, w, w, parentOfw);

        w.left = v.right; // v.right is y3 in Figure 43.6
        v.right = w;
      }
      else if (w.left == v && v.right == u) {
        // Case 1.2: v < u < w, Restructure and recolor nodes
        restructureRecolor(v, u, w, w, parentOfw);
        v.right = u.left;
        w.left = u.right;
        u.left = v;
        u.right = w;
      }
      else if (w.right == v && v.right == u) {
        // Case 1.3: w < v < u, Restructure and recolor nodes
        restructureRecolor(w, v, u, w, parentOfw);
        w.right = v.left;
        v.left = w;
      }
      else {
        // Case 1.4: w < u < v, Restructure and recolor nodes
        restructureRecolor(w, u, v, w, parentOfw);
        w.right = u.left;
        v.left = u.right;
        u.left = w;
        u.right = v;
      }
    }
    else { // Case 2: v's sibling x is red
      // Recolor nodes
      w.setRed();
      u.setRed();
      ((RBTreeNode<E>)(w.left)).setBlack();
      ((RBTreeNode<E>)(w.right)).setBlack();

      if (w == root) {
        w.setBlack();
      }
      else if (((RBTreeNode<E>)parentOfw).isRed()) {
        // Propagate along the path to fix new double-red violation
        u = w;
        v = (RBTreeNode<E>)parentOfw;
        fixDoubleRed(u, v, path, i - 2); // i – 2 propagates upward
      }
    }
  }

  /** Connect b with parentOfw and recolor a, b, c for a < b < c */
  private void restructureRecolor(RBTreeNode<E> a, RBTreeNode<E> b,
      RBTreeNode<E> c, RBTreeNode<E> w, RBTreeNode<E> parentOfw) {
    if (parentOfw == null)
      root = b;
    else if (parentOfw.left == w)
      parentOfw.left = b;
    else
      parentOfw.right = b;

    b.setBlack(); // b becomes the root in the subtree
    a.setRed(); // a becomes the left child of b
    c.setRed(); // c becomes the right child of b
  }

  @Override /** Delete an element from the RBTree.
    * Return true if the element is deleted successfully
    * Return false if the element is not in the tree */
  public boolean delete(E e) {
    // Locate the node to be deleted
    TreeNode<E> current = root;
    while (current != null) {
      if (e.compareTo(current.element) < 0) {
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        current = current.right;
      }
      else
        break; // Element is in the tree pointed by current
    }

    if (current == null)
      return false; // Element is not in the tree

    java.util.ArrayList<TreeNode<E>> path;

    // current node is an internal node
    if (current.left != null && current.right != null) {
      // Locate the rightmost node in the left subtree of current
      TreeNode<E> rightMost = current.left;
      while (rightMost.right != null) {
        rightMost = rightMost.right; // Keep going to the right
      }

      path = path(rightMost.element); // Get path before replacement

      // Replace the element in current by the element in rightMost
      current.element = rightMost.element;
    }
    else
      path = path(e); // Get path to current node

    // Delete the last node in the path and propagate if needed
    deleteLastNodeInPath(path);

    size--; // After one element deleted
    return true; // Element deleted
  }

  /** Delete the last node from the path. */
  public void deleteLastNodeInPath(ArrayList<TreeNode<E>> path) {
    int i = path.size() - 1; // Index to the node in the path
    // u is the last node in the path
    RBTreeNode<E> u = (RBTreeNode<E>)(path.get(i));
    RBTreeNode<E> parentOfu = (u == root) ? null :
      (RBTreeNode<E>)(path.get(i - 1));
    RBTreeNode<E> grandparentOfu = (parentOfu == null ||
      parentOfu == root) ? null :
      (RBTreeNode<E>)(path.get(i - 2));
    RBTreeNode<E> childOfu = (u.left == null) ?
      (RBTreeNode<E>)(u.right) : (RBTreeNode<E>)(u.left);

    // Delete node u. Connect childOfu with parentOfu
    connectNewParent(parentOfu, u, childOfu);

    // Recolor the nodes and fix double black if needed
    if (childOfu == root || u.isRed())
      return; // Done if childOfu is root or if u is red
    else if (childOfu != null && childOfu.isRed())
      childOfu.setBlack(); // Set it black, done
    else // u is black, childOfu is null or black
      // Fix double black on parentOfu
      fixDoubleBlack(grandparentOfu, parentOfu, childOfu, path, i);
    }

  /** Fix the double-black problem at node parent */
  private void fixDoubleBlack(
      RBTreeNode<E> grandparent, RBTreeNode<E> parent,
      RBTreeNode<E> db, ArrayList<TreeNode<E>> path, int i) {
    // Obtain y, y1, and y2
    RBTreeNode<E> y = (parent.right == db) ?
      (RBTreeNode<E>)(parent.left) : (RBTreeNode<E>)(parent.right);
    RBTreeNode<E> y1 = (RBTreeNode<E>)(y.left);
    RBTreeNode<E> y2 = (RBTreeNode<E>)(y.right);

    if (y.isBlack() && y1 != null && y1.isRed()) {
      if (parent.right == db) {
        // Case 1.1: y is a left black sibling and y1 is red
        connectNewParent(grandparent, parent, y);
        recolor(parent, y, y1); // Adjust colors

        // Adjust child links
        parent.left = y.right;
```

```
          y.right = parent;
        }
        else {
          // Case 1.3: y is a right black sibling and y1 is red
          connectNewParent(grandparent, parent, y1);
          recolor(parent, y1, y); // Adjust colors

          // Adjust child links
          parent.right = y1.left;
          y.left = y1.right;
          y1.left = parent;
          y1.right = y;
        }
      }
      else if (y.isBlack() && y2 != null && y2.isRed()) {
        if (parent.right == db) {
          // Case 1.2: y is a left black sibling and y2 is red
          connectNewParent(grandparent, parent, y2);
          recolor(parent, y2, y); // Adjust colors

          // Adjust child links
          y.right = y2.left;
          parent.left = y2.right;
          y2.left = y;
          y2.right = parent;
         }
        else {
          // Case 1.4: y is a right black sibling and y2 is red
          connectNewParent(grandparent, parent, y);
          recolor(parent, y, y2); // Adjust colors

          // Adjust child links
          y.left = parent;
          parent.right = y1;
        }
      }
      else if (y.isBlack()) {
        // Case 2: y is black and y's children are black or null
        y.setRed(); // Change y to red
        if (parent.isRed())
          parent.setBlack(); // Done
        else if (parent != root) {
          // Propagate double black to the parent node
          // Fix new appearance of double black recursively
          db = parent;
          parent = grandparent;
          grandparent =
            (i >= 3) ? (RBTreeNode<E>)(path.get(i - 3)) : null;
          fixDoubleBlack(grandparent, parent, db, path, i - 1);
        }
      }
      else { // y.isRed()
        if (parent.right == db) {
          // Case 3.1: y is a left red child of parent
          parent.left = y2;
          y.right = parent;
        }
        else {
          // Case 3.2: y is a right red child of parent
          parent.right = y.left;
            y.left = parent;
      }

      parent.setRed(); // Color parent red
      y.setBlack(); // Color y black
      connectNewParent(grandparent, parent, y); // y is new parent
      fixDoubleBlack(y, parent, db, path, i - 1);
    }
  }

  /** Recolor parent, newParent, and c. Case 1 removal */
  private void recolor(RBTreeNode<E> parent,
      RBTreeNode<E> newParent, RBTreeNode<E> c) {
    // Retain the parent's color for newParent
    if (parent.isRed())
      newParent.setRed();
    else
      newParent.setBlack();

    // c and parent become the children of newParent; set them black
    parent.setBlack();
    c.setBlack();
  }

  /** Connect newParent with grandParent */
  private void connectNewParent(RBTreeNode<E> grandparent,
      RBTreeNode<E> parent, RBTreeNode<E> newParent) {
    if (parent == root) {
      root = newParent;
      if (root != null)
        newParent.setBlack();
    }
    else if (grandparent.left == parent)
      grandparent.left = newParent;
    else
      grandparent.right = newParent;
  }

  @Override /** Preorder traversal from a subtree */
  protected void preorder(TreeNode<E> root) {
    if (root == null) return;
    System.out.print(root.element +
      (((RBTreeNode<E>)root).isRed() ? " (red) " : " (black) "));
    preorder(root.left);
    preorder(root.right);
  }

  /** RBTreeNode is TreeNode plus color indicator */
  protected static class RBTreeNode<E extends Comparable<E>> extends
      BST.TreeNode<E> {
    private boolean red = true; // Indicate node color

    public RBTreeNode(E e) {
      super(e);
    }

    public boolean isRed() {
      return red;
    }

    public boolean isBlack() {
      return !red;
```

```
342      }
343
344      public void setBlack() {
345        red = false;
346      }
347
348      public void setRed() {
349        red = true;
350      }
351
352      int blackHeight;
353    }
354  }
```

The `RBTree` class extends `BST`. Like the `BST` class, the `RBTree` class has a no-arg constructor that constructs an empty `RBTree` (lines 5–6) and a constructor that creates an initial `RBTree` from an array of elements (lines 9–11).

The `createNewNode()` method defined in the `BST` class creates a `TreeNode`. This method is overridden to return an `RBTreeNode` (lines 14–16). This method is invoked in the insert method in `BST` to create a node.

The `insert` method in `RBTree` is overridden in lines 20–29. The method first invokes the `insert` method in `BST`, then invokes `ensureRBTree(e)` (line 25) to ensure that tree is still a red-black tree after inserting a new element.

The `ensureRBTree(E e)` method first obtains the path of nodes that lead to element `e` from the root (line 34). It obtains `u` and `v` (the parent of `u`) from the path. If `u` is the root, color `u` black (lines 47–48). If `v` is red, invoke `fixDoubleRed` to fix the double red on both `u` and `v` (lines 49–50).

The `fixDoubleRed(u, v, path, i)` method fixes the double-red violation at node `u`. The method first obtains `w` (the grandparent of `u` from the path) (line 57), `parentOfw` if exists (lines 58–59), and `x` (the sibling of `v`) (lines 62–63). If `x` is `null` or black, consider four subcases to fix the double-red violation (lines 67–96). If x is red, color `w` and `u` red and color `w`'s two children black (lines 101–104). If `w` is the root, color `w` black (lines 104–106). Otherwise, propagate along the path to fix the new double-red violation (lines 109–111).

The `delete(E e)` method in `RBTree` is overridden in lines 134–174. The method locates the node that contains `e` (lines 136–146). If the node is null, no element is found (lines 148–149). The method considers two cases:

- If the node is internal, find the rightmost node in its left subtree (lines 156–159). Obtain a path from the root to the rightmost node (line 161), and replace the element in the node with the element in the rightmost node (line 164).
- If the node is external, obtain the path from the root to the node (line 167).

The last node in the path is the node to be deleted. Invoke `deleteLastNodeInPath(path)` to delete it and ensure the tree is a red-black after the node is deleted (line 170).

The `deleteLastNodeInPath(path)` method first obtains `u`, `parentOfu`, `grandparendOfu`, and `childOfu` (lines 180–187). `u` is the last node in the path. Connect `childOfu` as a child of `parentOfu` (line 190). This in effect deletes `u` from the tree. Consider three cases:

- If `childOfu` is the root or `childOfu` is red, we are done (lines 193–194).
- Otherwise, if `childOfu` is red, color it black (lines 195–196).
- Otherwise, invoke `fixDoubleBlack` to fix the double-black problem on `childOfu` (line 199).

The `fixDoubleBlack` method first obtains `y`, `y1`, and `y2` (lines 207–210). `y` is the sibling of the first double-black node, and `y1` and `y2` are the left and right children of `y`. Consider three cases:

- If `y` is black and `y1` or `y2` is red, fix the double-black problem for Case 1 (lines 213–255).
- Otherwise, if `y` is black, fix the double-black problem for Case 2 by recoloring the nodes. If parent is black and not a root, propagate double black to parent and recursively invoke `fixDoubleBlack` (lines 264–268).
- Otherwise, `y` is red. In this case, adjust the nodes to make parent the child of y (lines 272–281). Invoke `fixDoubleBlack` with the adjusted nodes (line 286) to fix the double-black problem.

The `preorder(TreeNode<E> root)` method is overridden to display the node colors (lines 319–325).

43.7 Testing the RBTree Class

This section gives a test program that uses the `RBTree` *class.*

Key Point

Listing 43.4 gives a test program. The program creates an `RBTree` initialized with an array of integers `34`, `3`, and `50` (lines 4–5), inserts elements in lines 10–22, and deletes elements in lines 25–46.

LISTING 43.4 TestRBTree.java

```
 1  public class TestRBTree {
 2    public static void main(String[] args) {
 3      // Create an RB tree
 4      RBTree<Integer> tree =
 5        new RBTree<Integer>(new Integer[]{34, 3, 50});
 6      printTree(tree);
 7
 8      tree.insert(20);
 9      printTree(tree);
10
11      tree.insert(15);
12      printTree(tree);
13
14      tree.insert(16);
15      printTree(tree);
16
17      tree.insert(25);
18      printTree(tree);
19
20      tree.insert(27);
21      printTree(tree);
22
23      tree.delete(50);
24      printTree(tree);
25
26      tree.delete(20);
27      printTree(tree);
28
29      tree.delete(15);
30      printTree(tree);
31
```

```java
    tree.delete(3);
    printTree(tree);

    tree.delete(25);
    printTree(tree);

    tree.delete(16);
    printTree(tree);

    tree.delete(34);
    printTree(tree);

    tree.delete(27);
    printTree(tree);
  }

  public static <E extends Comparable<E>>
    void printTree(BST <E> tree) {
    // Traverse tree
    System.out.print("\nInorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```

```
Inorder (sorted): 3 34 50
Postorder: 3 50 34
Preorder: 34 (black) 3 (red) 50 (red)
The number of nodes is 3

Inorder (sorted): 3 20 34 50
Postorder: 20 3 50 34
Preorder: 34 (black) 3 (black) 20 (red) 50 (black)
The number of nodes is 4

Inorder (sorted): 3 15 20 34 50
Postorder: 3 20 15 50 34
Preorder: 34 (black) 15 (black) 3 (red) 20 (red) 50 (black)
The number of nodes is 5

Inorder (sorted): 3 15 16 20 34 50
Postorder: 3 16 20 15 50 34
Preorder: 34 (black) 15 (red) 3 (black) 20 (black) 16 (red) 50 (black)
The number of nodes is 6

Inorder (sorted): 3 15 16 20 25 34 50
Postorder: 3 16 25 20 15 50 34
Preorder: 34 (black) 15 (red) 3 (black) 20 (black) 16 (red) 25 (red)
  50 (black)
The number of nodes is 7

Inorder (sorted): 3 15 16 20 25 27 34 50
Postorder: 3 16 15 27 25 50 34 20
```

```
Preorder: 20 (black) 15 (red) 3 (black) 16 (black) 34 (red) 25 (black)
  27 (red) 50 (black)
The number of nodes is 8

Inorder (sorted): 3 15 16 20 25 27 34
Postorder: 3 16 15 25 34 27 20
Preorder: 20 (black) 15 (red) 3 (black) 16 (black) 27 (red)
  25 (black) 34 (black)
The number of nodes is 7

Inorder (sorted): 3 15 16 25 27 34
Postorder: 3 15 25 34 27 16
Preorder: 16 (black) 15 (black) 3 (red) 27 (red) 25 (black) 34 (black)
The number of nodes is 6

Inorder (sorted): 3 16 25 27 34
Postorder: 3 25 34 27 16
Preorder: 16 (black) 3 (black) 27 (red) 25 (black) 34 (black)
The number of nodes is 5

Inorder (sorted): 16 25 27 34
Postorder: 25 16 34 27
Preorder: 27 (black) 16 (black) 25 (red) 34 (black)
The number of nodes is 4

Inorder (sorted): 16 27 34
Postorder: 16 34 27
Preorder: 27 (black) 16 (black) 34 (black)
The number of nodes is 3

Inorder (sorted): 27 34
Postorder: 34 27
Preorder: 27 (black) 34 (red)
The number of nodes is 2

Inorder (sorted): 27
Postorder: 27
Preorder: 27 (black)
The number of nodes is 1

Inorder (sorted):
Postorder:
Preorder:
The number of nodes is 0
```

Figure 43.14 shows how the tree evolves as elements are added to it, and Figure 43.28 shows how the tree evolves as elements are deleted from it.

43.8 Performance of the `RBTree` Class

This search, insertion, and deletion operations take O(logn) time in a red-black tree.

Key
Point

The search, insertion, and deletion times in a red-black tree depend on the height of the tree. A red-black tree corresponds to a 2–4 tree. When you convert a node in a 2–4 tree to red-black tree nodes, you get one black node and zero, one, or two red nodes as its children, depending on whether the original node is a 2-node, 3-node, or 4-node. So, the height of a red-black tree

TABLE 43.1 Time Complexities for Methods in `RBTree`, `AVLTree`, and `Tree234`

Mehtods	*Red-Black Tree*	*AVL Tree*	*2-4 Tree*
`search (e: E)`	$O(\log n)$	$O(\log n)$	$O(\log n)$
`insert (e: E)`	$O(\log n)$	$O(\log n)$	$O(\log n)$
`delete (e: E)`	$O(\log n)$	$O(\log n)$	$O(\log n)$
`getSize()`	$O(1)$	$O(1)$	$O(1)$
`isEmpty()`	$O(1)$	$O(1)$	$O(1)$

is at most as twice that of its corresponding 2–4 tree. Since the height of a 2–4 tree is log n, the height of a red-black tree is 2log n.

A red-black tree has the same time complexity as an AVL tree, as shown in Table 43.1. In general, a red-black is more efficient than an AVL tree, because a red-black tree requires only one-time restructuring of the nodes for insert and delete operations.

A red-black tree has the same time complexity as a 2–4 tree, as shown in Table 43.1. In general, a red-black is more efficient than a 2–4 tree for two reasons:

1. A red-black tree requires only one-time restructuring of the nodes for insert and delete operations. However, a 2–4 tree may require many splits for an insert operation and fusion for a delete operation.
2. A red-black tree is a binary search tree. A binary tree can be implemented more space efficiently than a 2–4 tree, because a node in a 2–4 tree has at most three elements and four children. Space is wasted for 2-nodes and 3-nodes in a 2–4 tree.

Listing 43.5 gives an empirical test of the performance of AVL trees, 2–4 trees, and red-black trees.

LISTING 43.5 TreePerformanceTest.java

```java
public class TreePerformanceTest {
  public static void main(String[] args) {
    final int TEST_SIZE = 500000; // Tree size used in the test

    // Create an AVL tree
    Tree<Integer> tree1 = new AVLTree<Integer>();
    System.out.println("AVL tree time: " +
      getTime(tree1, TEST_SIZE) + " milliseconds");

    // Create a 2-4 tree
    Tree<Integer> tree2 = new Tree24<Integer>();
    System.out.println("2-4 tree time: "
      + getTime(tree2, TEST_SIZE) + " milliseconds");

    // Create a red-black tree
    Tree<Integer> tree3 = new RBTree<Integer>();
    System.out.println("RB tree time: "
      + getTime(tree3, TEST_SIZE) + " milliseconds");
  }

  public static long getTime(Tree<Integer> tree, int testSize) {
    long startTime = System.currentTimeMillis(); // Start time

    // Create a list to store distinct integers
    java.util.List<Integer> list = new java.util.ArrayList<Integer>();
26       for (int i = 0; i < testSize; i++)
27         list.add(i);
28
29       java.util.Collections.shuffle(list); // Shuffle the list
30
31       // Insert elements in the list to the tree
32       for (int i = 0; i < testSize; i++)
33         tree.insert(list.get(i));
34
35       java.util.Collections.shuffle(list); // Shuffle the list
36
37       // Delete elements in the list from the tree
38       for (int i = 0; i < testSize; i++)
39         tree.delete(list.get(i));
40
41       // Return elapse time
42       return System.currentTimeMillis() - startTime;
43     }
44   }
```

```
AVL tree time: 7609 milliseconds
2-4 tree time: 8594 milliseconds
RB tree time: 5515 milliseconds
```

The `getTestTime` method creates a list of distinct integers from `0` to `testSize – 1` (lines 25–27), shuffles the list (line 29), adds the elements from the list to a tree (lines 32–33), shuffles the list again (line 35), removes the elements from the tree (lines 38–39), and finally returns the execution time (line 42).

The program creates an AVL (line 6), a 2-4 tree (line 11), and a red-black tree (line 16). The program obtains the execution time for adding and removing `500000` elements in the three trees.

As you see, the red-black tree performs the best, followed by the AVL tree.

Note

The `java.util.TreeSet` class in the Java API is implemented using a red-black tree. Each entry in the set is stored in the tree. Since the `search`, `insert`, and `delete` methods in a red-black tree take *O*(log *n*) time, the `get`, `add`, `remove`, and `contains` methods in `java.util.TreeSet` take *O*(log *n*) time.

Note

The `java.util.TreeMap` class in the Java API is implemented using a red-black tree. Each entry in the map is stored in the tree. The order of the entries is determined by their keys. Since the `search`, `insert`, and `delete` methods in a red-black tree take *O*(log *n*) time, the `get`, `put`, `remove`, and `containsKey` methods in `java.util.TreeMap` take *O*(log *n*) time.

KEY TERMS

black depth	43-2
double-black violation	43-11
double-red violation	43-7
external node	43-9
red-black tree	43-2

CHAPTER SUMMARY

1. A red-black tree is a binary search tree, derived from a *2-4 tree*. A red-black tree corresponds to a 2-4 tree. You can convert a red-black tree to a 2-4 tree or vice versa.

2. In a red-black tree, each node is colored red or black. The root is always black. Two adjacent nodes cannot be both red. All external nodes have the same black depth.

3. Since a red-black tree is a binary search tree, the `RBTree` class extends the `BST` class.

4. Searching an element in a red-black tree is the same as in binary search tree, since a red-black tree is a binary search tree.

5. A new element is always inserted as a leaf node. If the new node is the root, color it black. Otherwise, color it red. If the parent of the new node is red, we have to fix the *double-red violation* by reassigning the color and/or restructuring the tree.

6. If a node to be deleted is internal, find the rightmost node in its left subtree. Replace the element in the node with the element in the rightmost node. Delete the rightmost node.

7. If the external node to be deleted is red, simply reconnect the parent node of the external node with the child node of the external node.

8. If the external node to be deleted is black, you need to consider several cases to ensure that black height for external nodes in the tree is maintained correctly.

9. The height of a red-black tree is O(logn). So, the time complexities for the `search`, `insert`, and `delete` methods are O(logn).

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

***43.1** (*red-black tree to 2-4 tree*) Write a program that converts a red-black tree to a 2-4 tree.

***43.2** (*2-4 tree to red-black tree*) Write a program that converts a red-black tree to a 2-4 tree.

*****43.3** (*red-black tree animation*) Write a GUI program that animates the red-black tree `insert`, `delete`, and `search` methods, as shown in Figure 43.6.

****43.4** (*Parent reference for `RBTree`*) Suppose that the `TreeNode` class defined in `BST` contains a reference to the node's parent, as shown in Exercise 26.17. Implement the `RBTree` class to support this change. Write a test program that adds numbers `1`, `2`, . . . , `100` to the tree and displays the paths for all leaf nodes.

CHAPTER

44

TESTING USING JUNIT

Objectives

- To know what JUnit is and how JUnit works (§44.2).
- To create and run a JUnit test class from the command window (§44.2).
- To create and run a JUnit test class from NetBeans (§44.3).
- To create and run a JUnit test class from Eclipse (§44.4).

44.1 Introduction

Key Point

JUnit is a tool for testing Java programs.

At the very beginning of this book in Section 2.16, we introduced software development process that includes requirements specification, analysis, design, implementation, testing, deployment, and maintenance. Testing is an important part of this process. This chapter introduces how to test Java classes using JUnit.

44.2 JUnit Basics

Key Point

To test a class, you need to write a test class and run it through JUnit to generate a report for the class.

JUnit is the de facto framework for testing Java programs. JUnit is a third-party open-source library packed in a jar file. The jar file contains a tool called *test runner*, which is used to run test programs. Suppose you have a class named `A`. To test this class, you write a test class named `ATest`. This test class, called a *test class*, contains the methods you write for testing class `A`. The test runner executes `ATest` to generate a test report, as shown in Figure 44.1.

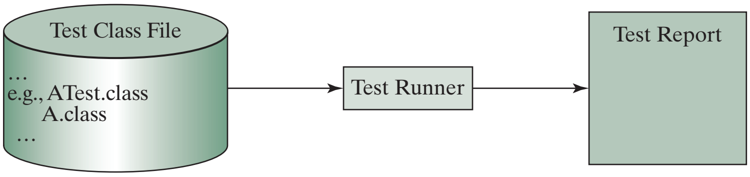


FIGURE 44.1 JUnit test runner executes the test class to generate a test report.

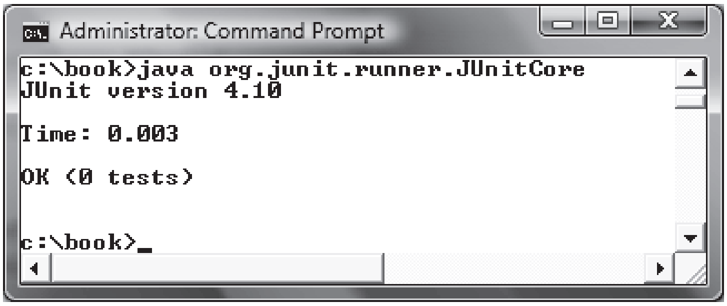
You will see how JUnit works from an example. To create the example, first you need to download JUnit from http://sourceforge.net/projects/junit/files/. At present, the latest version is junit-4.10.jar. Download this file to c:\book\lib and add it to the classpath environment variable as follows:

```
set classpath=.;%classpath%;c:\book\lib\junit-4.10.jar
```

To test if this environment variable is set correctly, open a new command window, and type the following command:

```
java org.junit.runner.JUnitCore
```

You should see the message displayed as shown in Figure 44.2.



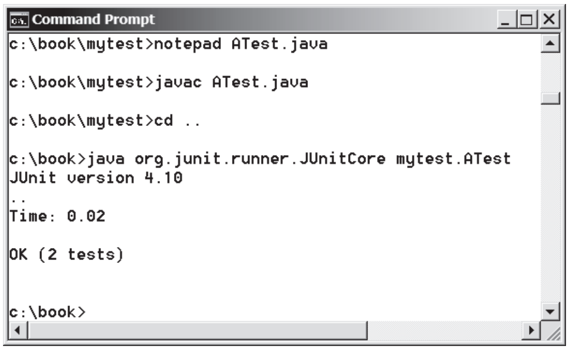
```
c:\book>java org.junit.runner.JUnitCore
JUnit version 4.10

Time: 0.003

OK (0 tests)


c:\book>_
```

FIGURE 44.2 The JUnit test runner displays the JUnit version.

To use JUnit, create a test class. By convention, if the class to be tested is named `A`, the test class should be named `ATest`. A simple template of a test class may look like this:

```
package mytest;

import org.junit.*;
import static org.junit.Assert.*;

public class ATest {
  @Test
  public void m1() {
    // Write a test method
  }

  @Test
  public void m2() {
    // Write another test method
  }

  @Before
  public void setUp() throws Exception {
    // Common objects used by test methods may be set up here
  }
}
```

This class should be placed in a directory under mytest. Suppose the class is placed under c:\book\mytest. You need to compile it from the mytest directory and run it from c:\book as shown in the following screen shot.



```
c:\book\mytest>notepad ATest.java

c:\book\mytest>javac ATest.java

c:\book\mytest>cd ..

c:\book>java org.junit.runner.JUnitCore mytest.ATest
JUnit version 4.10
..
Time: 0.02

OK (2 tests)


c:\book>
```

Note that the command to run the test from the console is:

```
java org.junit.runner.JUnitCore mytest.ATest
```

When this command is executed, `JUnitCore` controls the execution of `ATest`. It first executes the `setUp()` method to set up the common objects used for the test, and then executes test methods `m1` and `m2` in this order. You may define multiple test methods if desirable.

The following methods can be used to implement a test method:

`assertTrue(booleanExpression)`
The method reports success if the booleanExpression evaluates true.

`assertEquals(Object, Object)`
The method reports success if the two objects are the same using the `equals` method.

`assertNull(Object)`
The method reports success if the object reference passed is `null`.

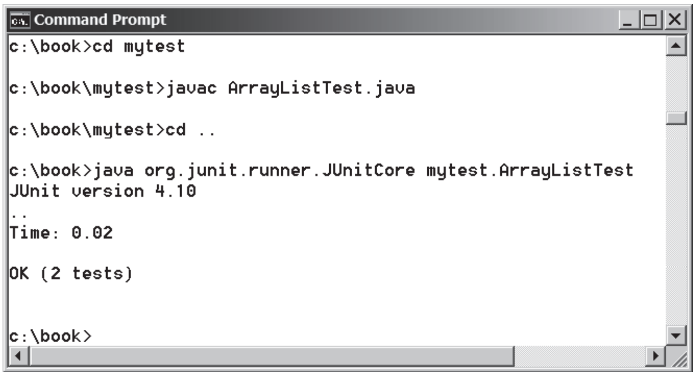
`fail(String)`
The method causes the test to fail and prints out the string.

Listing 44.1 is an example of a test class for testing `java.util.ArrayList`.

LISTING 44.1 `ArrayListTest.java`

```
package mytest;

import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class ArrayListTest {
  private ArrayList<String> list = new ArrayList<String>();

  @Before
  public void setUp() throws Exception {
  }

  @Test
  public void testInsertion() {
    list.add("Beijing");
    assertEquals("Beijing", list.get(0));
    list.add("Shanghai");
    list.add("Hongkong");
    assertEquals("Hongkong", list.get(list.size() – 1));
  }

  @Test
  public void testDeletion() {
    list.clear();
    assertTrue(list.isEmpty());

    list.add("A");
    list.add("B");
    list.add("C");
    list.remove("B");
    assertEquals(2, list.size());
  }
}
```

A test run of the program is shown in Figure 44.3. Note that you have to first compile ArrayListTest.java. The ArrayListTest class is placed in the `mytest` package. So you should place ArrayListTest.java in the directory named `mytest`.



```
c:\book>cd mytest

c:\book\mytest>javac ArrayListTest.java

c:\book\mytest>cd ..

c:\book>java org.junit.runner.JUnitCore mytest.ArrayListTest
JUnit version 4.10
..
Time: 0.02

OK (2 tests)


c:\book>
```

FIGURE 44.3 The test report is displayed from running ArrayListTest.

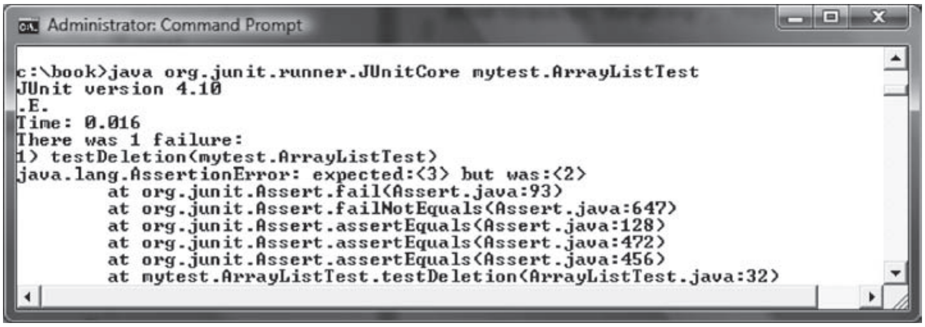
No errors are reported in this JUnit run. If you mistakenly change

```
assertEquals(2, list.size());
```

in line 32 to

```
assertEquals(3, list.size());
```

Run `ArrayListTest` now. You will see an error reported as shown in Figure 44.4.



```
c:\book>java org.junit.runner.JUnitCore mytest.ArrayListTest
JUnit version 4.10
.E.
Time: 0.016
There was 1 failure:
1) testDeletion(mytest.ArrayListTest)
java.lang.AssertionError: expected:<3> but was:<2>
        at org.junit.Assert.fail(Assert.java:93)
        at org.junit.Assert.failNotEquals(Assert.java:647)
        at org.junit.Assert.assertEquals(Assert.java:128)
        at org.junit.Assert.assertEquals(Assert.java:472)
        at org.junit.Assert.assertEquals(Assert.java:456)
        at mytest.ArrayListTest.testDeletion(ArrayListTest.java:32)
```

FIGURE 44.4 The test report reports an error.

You can define any number of test methods. In this example, the two test methods `testInsertion` and `testDeletion` are defined. JUnit executes `testInsertion` and `testDeletion` in this order.

Note

The test class must be placed in a named package such as `mytest` in this example. The JUnit will not work if the test class is placed a default package.

Listing 44.2 gives a test class for testing the `Loan` class in Listing 10.2. For convenience, we create Loan.java in the same directory with LoanTest.java. The `Loan` class is shown in Listing 44.3.

LISTING 44.2 LoanTest.java

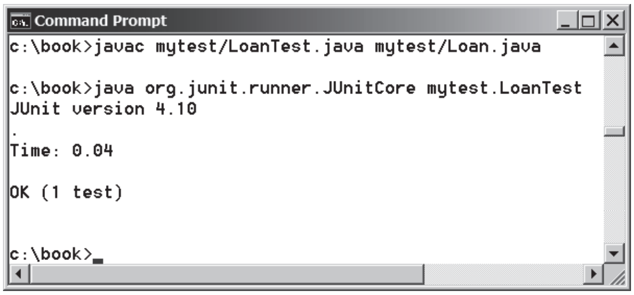
```java
package mytest;

import org.junit.*;
import static org.junit.Assert.*;

public class LoanTest {
  @Before
  public void setUp() throws Exception {
  }

  @Test
  public void testPaymentMethods() {
    double annualInterestRate = 2.5;
    int numberOfYears = 5;
    double loanAmount = 1000;
    Loan loan = new Loan(annualInterestRate, numberOfYears,
      loanAmount);

    assertTrue(loan.getMonthlyPayment() ==
      getMonthlyPayment(annualInterestRate, numberOfYears,
      loanAmount));
    assertTrue(loan.getTotalPayment() ==
      getTotalPayment(annualInterestRate, numberOfYears,
      loanAmount));
  }

  /** Find monthly payment */
  private double getMonthlyPayment(double annualInterestRate,
      int numberOfYears, double loanAmount) {
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 –
      (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
    return monthlyPayment;
  }

  /** Find total payment */
  public double getTotalPayment(double annualInterestRate,
      int numberOfYears, double loanAmount) {
    return getMonthlyPayment(annualInterestRate, numberOfYears,
      loanAmount) * numberOfYears * 12;
  }
}
```

LISTING 44.3 Loan.java

```
package mytest;

public class Loan {
  private double annualInterestRate;
  private int numberOfYears;
  private double loanAmount;
  private java.util.Date loanDate;

  /** Default constructor */
  public Loan() {
    this(2.5, 1, 1000);
  }

  /** Construct a loan with specified annual interest rate,
      number of years, and loan amount
    */
  public Loan(double annualInterestRate, int numberOfYears,
      double loanAmount) {
    this.annualInterestRate = annualInterestRate;
    this.numberOfYears = numberOfYears;
    this.loanAmount = loanAmount;
    loanDate = new java.util.Date();
  }

  /** Return annualInterestRate */
  public double getAnnualInterestRate() {
    return annualInterestRate;
  }

  /** Set a new annualInterestRate */
  public void setAnnualInterestRate(double annualInterestRate) {
    this.annualInterestRate = annualInterestRate;
  }
```

```java

  /** Return numberOfYears */
  public int getNumberOfYears() {
    return numberOfYears;
  }

  /** Set a new numberOfYears */
  public void setNumberOfYears(int numberOfYears) {
    this.numberOfYears = numberOfYears;
  }

  /** Return loanAmount */
  public double getLoanAmount() {
    return loanAmount;
  }

  /** Set a newloanAmount */
  public void setLoanAmount(double loanAmount) {
    this.loanAmount = loanAmount;
  }

  /** Find monthly payment */
  public double getMonthlyPayment() {
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
      (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
    return monthlyPayment;
  }

  /** Find total payment */
  public double getTotalPayment() {
    double totalPayment = getMonthlyPayment() * numberOfYears * 12;
    return totalPayment;
  }

  /** Return loan date */
  public java.util.Date getLoanDate() {
    return loanDate;
  }
}
```

The `testPaymentMethods()` in `LoanTest` creates an instance of `Loan` (line 16–17) and tests whether `loan.getMonthlyPayment()` returns the same value as `getMonthlyPayment(annualInterestRate, numberOfYears, loanAmount)`. The latter method is defined in the `LoanTest` class (lines 28–34).



```
c:\book>javac mytest/LoanTest.java mytest/Loan.java

c:\book>java org.junit.runner.JUnitCore mytest.LoanTest
JUnit version 4.10
.
Time: 0.04

OK (1 test)


c:\book>_
```

FIGURE 44.5 The JUnit test runner executes LoanTest and reports no errors.

The `testPaymentMethods()` also tests whether `loan.getTotalPayment()` returns the same value as `getTotalPayment(annualInterestRate, numberOfYears, loanAmount)`. The latter method is defined in the `LoanTest` class (lines 37–41).

A sample run of the program is shown in Figure 44.5.

Check Point

44.2.1 What is JUnit?

44.2.2 What is a JUnit test runner?

44.2.3 What is a test class? How do you create a test class?

44.2.4 How do you use the `assertTrue` method?

44.2.5 How do you use the `assertEquals` method?

44.3 Using JUnit from NetBeans

Key Point

JUnit is intergrated with NetBeans. Using NetBeans, the test program can be automatically generated and the test process can be automated.

An IDE such as NetBeans and Eclipse can greatly simplify the process for creating and running test classes. This section introduces using JUnit from NetBeans and the next section introduces using JUnit from Eclipse.

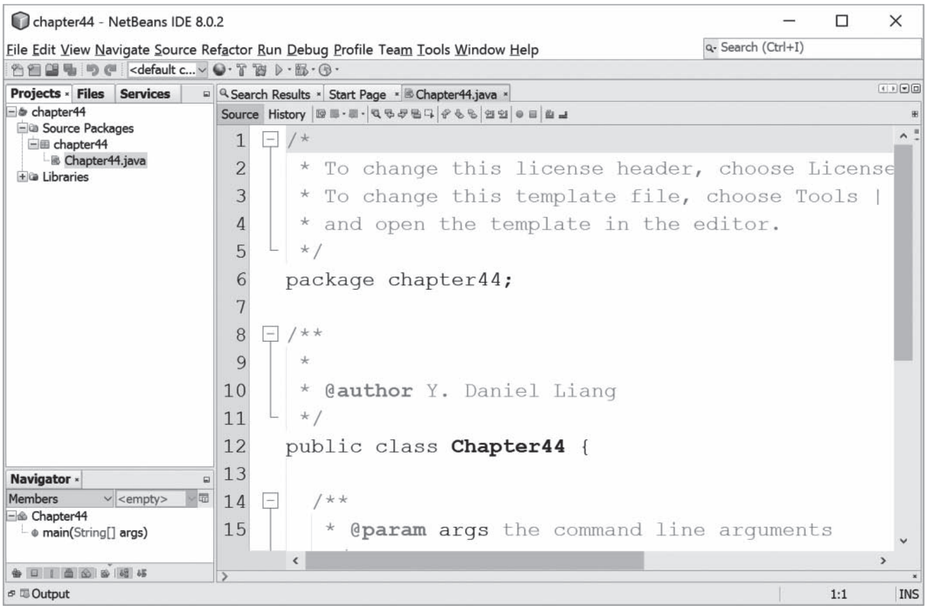
If you are not familiar with NetBeans, see Supplement II.B. Assume you have installed NetBeans 8 or higher. Create a project named `chapter44` as follows:

Step 1: Choose *File*, *New Project* to display the New Project dialog box.

Step 2: Choose Java in the Categories section and Java Application in the Projects section. Click *Next* to display the New Java Application dialog box.

Step 3: Enter `chapter44` as the Project Name and `c:\book` as Project Location. Click *Finish* to create the project as shown in Figure 44.6.

To demonstrate how to create a test class, we first create a class to be tested. Let the class be `Loan` from Listing 10.2. Here are the steps to create the `Loan` class under chapter44.


chapter44 - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default c...
Projects
Files
Services
chapter44
Source Packages
chapter44
Chapter44.java
Libraries
Search Results
Start Page
Chapter44.java
Source
History
Navigator
Members
<empty>
Chapter44
main(String[] args)
Output
1:1
INS

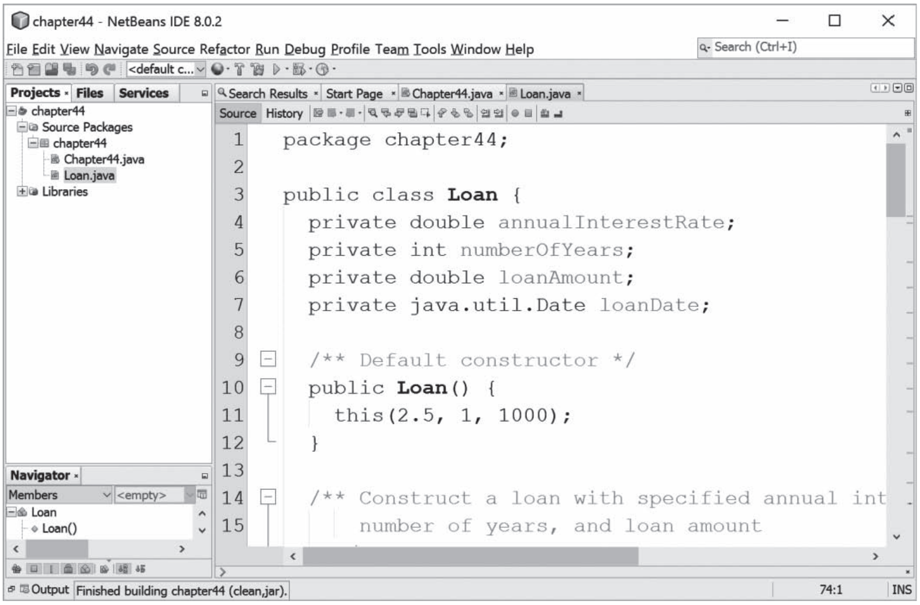
```
/*
 * To change this license header, choose License
 * To change this template file, choose Tools |
 * and open the template in the editor.
 */
package chapter44;

/**
 *
 * @author Y. Daniel Liang
 */
public class Chapter44 {

    /**
     * @param args the command line arguments
```

Figure 44.6 A new project named chapter44 is created.

Step 1: Right-click the project node chapter44 and choose *New*, *Java Class* to display the New Java Class dialog box.

Step 2: Enter `Loan` as Class Name and `chapter44` in the Package field and click *Finish* to create the class.

Step 3: Copy the code in Listing 10.2 to the `Loan` class and make sure the first line is `package chapter44`, as shown in Figure 44.7.


chapter44 - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default c...
Projects
Files
Services
chapter44
Source Packages
chapter44
Chapter44.java
Loan.java
Libraries
Search Results
Start Page
Chapter44.java
Loan.java
Source
History
Navigator
Members
<empty>
Loan
Loan()
Output
74:1
INS

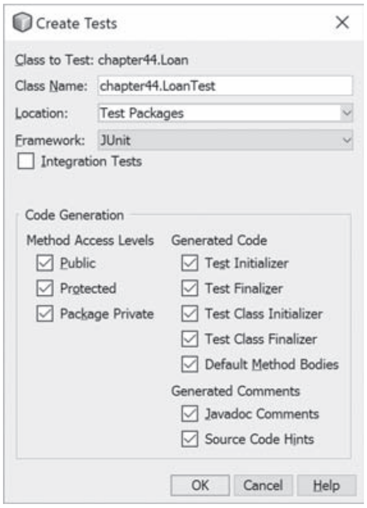
```
package chapter44;

public class Loan {
  private double annualInterestRate;
  private int numberOfYears;
  private double loanAmount;
  private java.util.Date loanDate;

  /** Default constructor */
  public Loan() {
    this(2.5, 1, 1000);
  }

  /** Construct a loan with specified annual int
      number of years, and loan amount
```

Finished building chapter44 (clean,jar).

FIGURE 44.7 The `Loan` class is created.

Now you can create a test class to test the `Loan` class as follows:

Step 1: Right-click `Loan.java` in the project to display a context menu and choose *Tools*, *Create/Update Test* to display the Create Test dialog box, as shown in Figure 44.8.


Create Tests
OK
Cancel
Help

Class to Test: chapter44.Loan

Class Name: chapter44.LoanTest

Location: Test Packages

Framework: JUnit

☐ Integration Tests

Code Generation

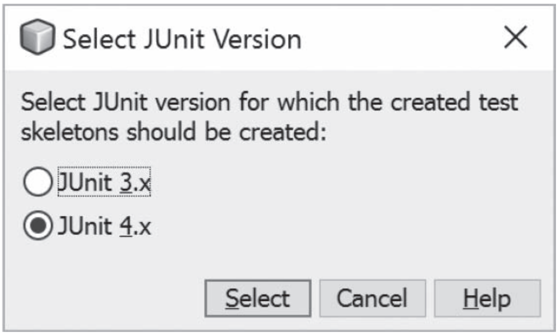
Method Access Levels
- ☑ Public
- ☑ Protected
- ☑ Package Private

Generated Code
- ☑ Test Initializer
- ☑ Test Finalizer
- ☑ Test Class Initializer
- ☑ Test Class Finalizer
- ☑ Default Method Bodies

Generated Comments
- ☑ Javadoc Comments
- ☑ Source Code Hints

FIGURE 44.8 The Create Tests dialog box creates a Test class.

Step 2: Click OK. You will see the Select JUnit version dialog box displayed as shown in Figure 44.9. Choose Junit 4.x. Click *OK* to generate a Test class named LoanTest as shown in Figure 44.10. Note that LoanTest.java is placed under the Test Packages node in the project.

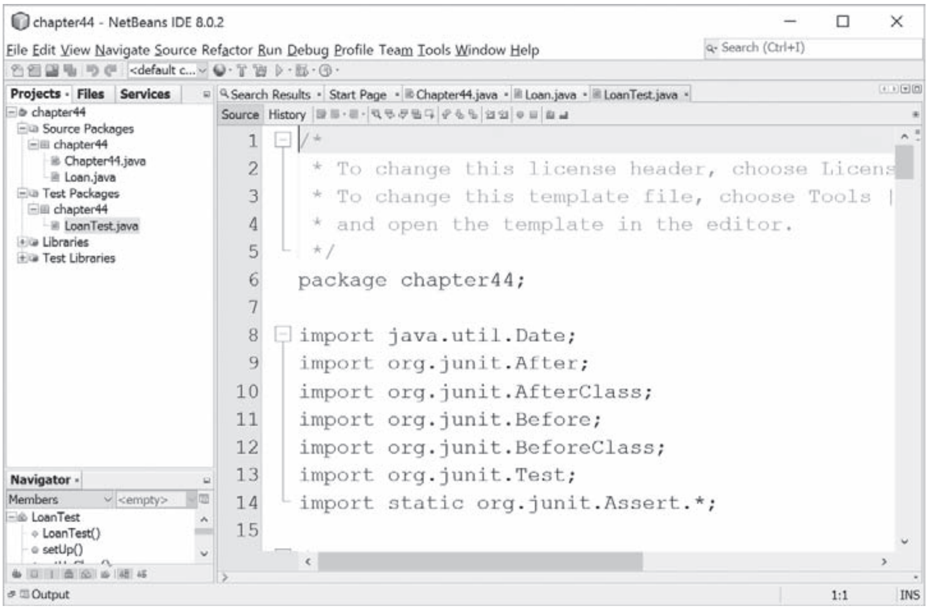

Select JUnit Version
Select
Cancel
Help

Select JUnit version for which the created test skeletons should be created:

○ JUnit 3.x

● JUnit 4.x

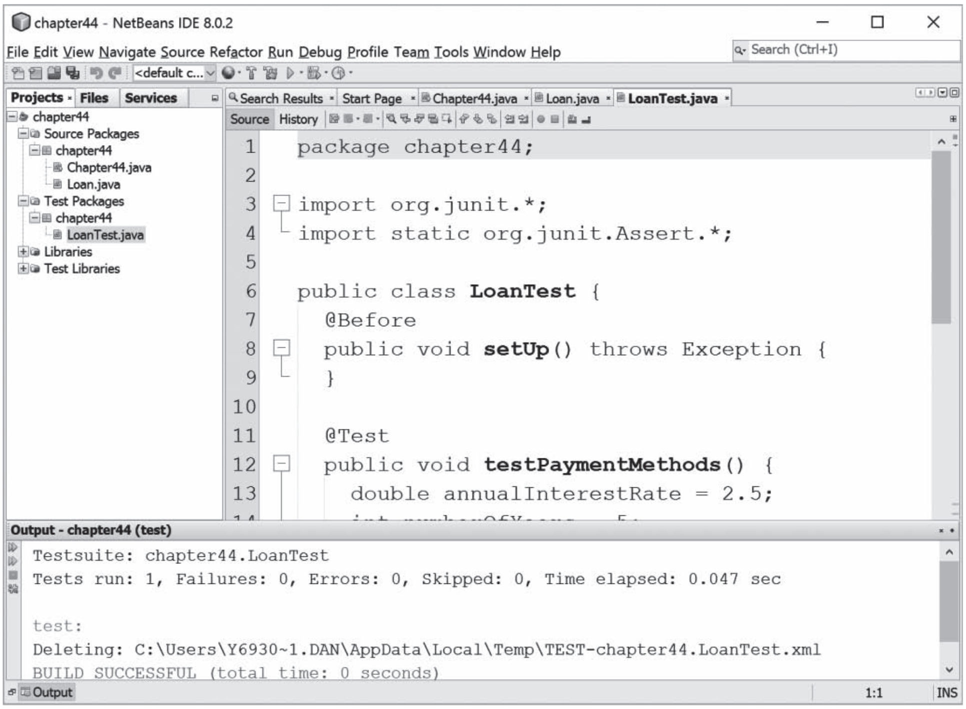
FIGURE 44.9 You should select JUnit 4.x framework to create test classes.


chapter44 - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default c...
Projects
Files
Services
chapter44
Source Packages
chapter44
Chapter44.java
Loan.java
Test Packages
chapter44
LoanTest.java
Libraries
Test Libraries
Search Results
Start Page
Chapter44.java
Loan.java
LoanTest.java
Source
History
Navigator
Members
<empty>
LoanTest
LoanTest()
setUp()
Output
1:1
INS

```
/*
 * To change this license header, choose Licens
 * To change this template file, choose Tools |
 * and open the template in the editor.
 */
package chapter44;

import java.util.Date;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

```

FIGURE 44.10 The LoanTest class is automatically generated.

You can now modify LoanTest by copying the code from Listing 44.2. Run LoanTest.java. You will see the test report as shown in Figure 44.11.


chapter44 - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default c...
Projects
Files
Services
chapter44
Source Packages
chapter44
Chapter44.java
Loan.java
Test Packages
chapter44
LoanTest.java
Libraries
Test Libraries
Search Results
Start Page
Chapter44.java
Loan.java
LoanTest.java
Source
History
Output - chapter44 (test)
Output
1:1
INS

```
package chapter44;

import org.junit.*;
import static org.junit.Assert.*;

public class LoanTest {
  @Before
  public void setUp() throws Exception {
  }

  @Test
  public void testPaymentMethods() {
    double annualInterestRate = 2.5;
```

```
Testsuite: chapter44.LoanTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.047 sec

test:
Deleting: C:\Users\Y6930~1.DAN\AppData\Local\Temp\TEST-chapter44.LoanTest.xml
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 44.11 The test report is displayed after the LoanTest class is executed.

44.4 Using JUnit from Eclipse

Key Point

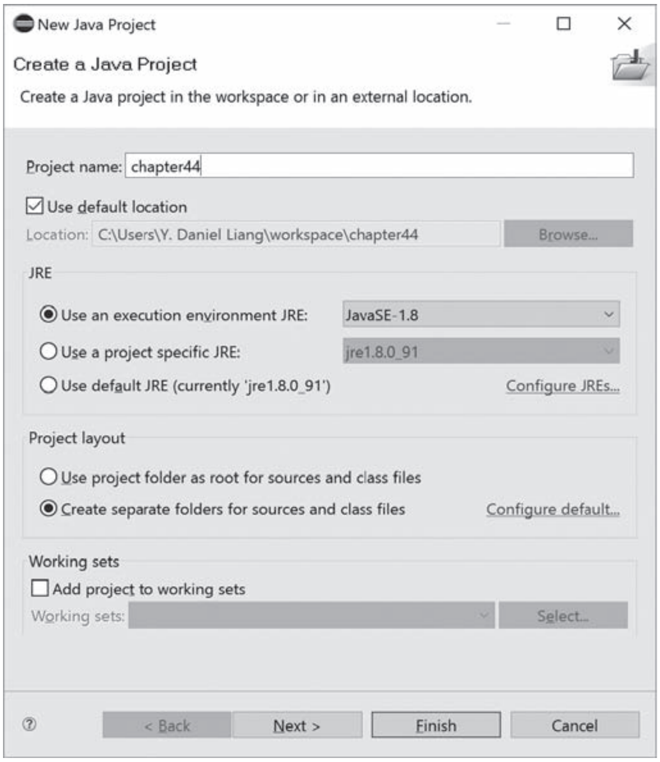
JUnit is intergrated with Eclipse. Using Eclipse, the test program can be automatically generated and the test process can be automated.

This section introduces using JUnit from Eclipse. If you are not familiar with Eclipse, see Supplement II.D. Assume you have installed Eclipse 4.5 or higher. Create a project named `chapter50` as follows:

Step 1: Choose *File*, *New Java Project* to display the New Java Project dialog box, as shown in Figure 44.12.

Step 2: Enter `chapter50` in the project name field and click *Finish* to create the project.

To demonstrate how to create a test class, we first create a class to be tested. Let the class be `Loan` from Listing 10.2. Here are the steps to create the `Loan` class under chapter44.



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: chapter44

☑ Use default location

Location: C:\Users\Y. Daniel Liang\workspace\chapter44

JRE

● Use an execution environment JRE: JavaSE-1.8

○ Use a project specific JRE: jre1.8.0_91

○ Use default JRE (currently 'jre1.8.0_91')

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files

Working sets

☐ Add project to working sets

Working sets:

FIGURE 44.12 The New Java Project dialog creates a new project.

Step 1: Right-click the project node chapter44 and choose *New*, *Class* to display the New Java Class dialog box, as shown in Figure 44.13.

Step 2: Enter `mytest` in the Package field and click *Finish* to create the class.

Step 3: Copy the code in Listing 10.2 to the `Loan` class and make sure the first line is `package mytest`, as shown in Figure 44.14.

Now you can create a test class to test the `Loan` class as follows:

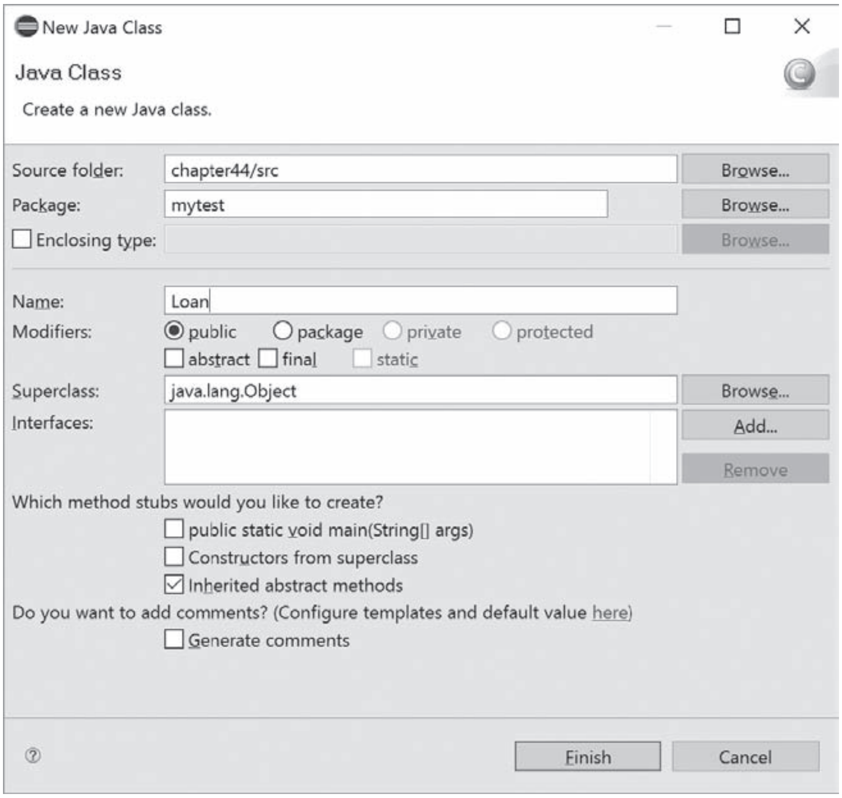
Step 1: Right-click `Loan.java` in the project to display a context menu and choose *New*, *JUnit Test Case* to display the New JUnit Test Case dialog box, as shown in Figure 44.15.

Step 2: Click *Finish*. You will see a dialog prompting you to add JUnit 4 to the project build path. Click *OK* to add it. Now a test class named LoanTest is created as shown in Figure 44.16.

You can now modify LoanTest by copying the code from Listing 44.2. Run LoanTest.java. You will see the test report as shown in Figure 44.17.

KEY TERMS

JUnit	44-2
JUnitCore	44-2
test class	44-2
test runner	44-2


New Java Class
Java Class
Create a new Java class.
Browse...
Browse...
Browse...
Browse...
Add...
Remove
Finish
Cancel

Source folder: chapter44/src

Package: mytest

☐ Enclosing type:

Name: Loan

Modifiers: ● public ○ package ○ private ○ protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

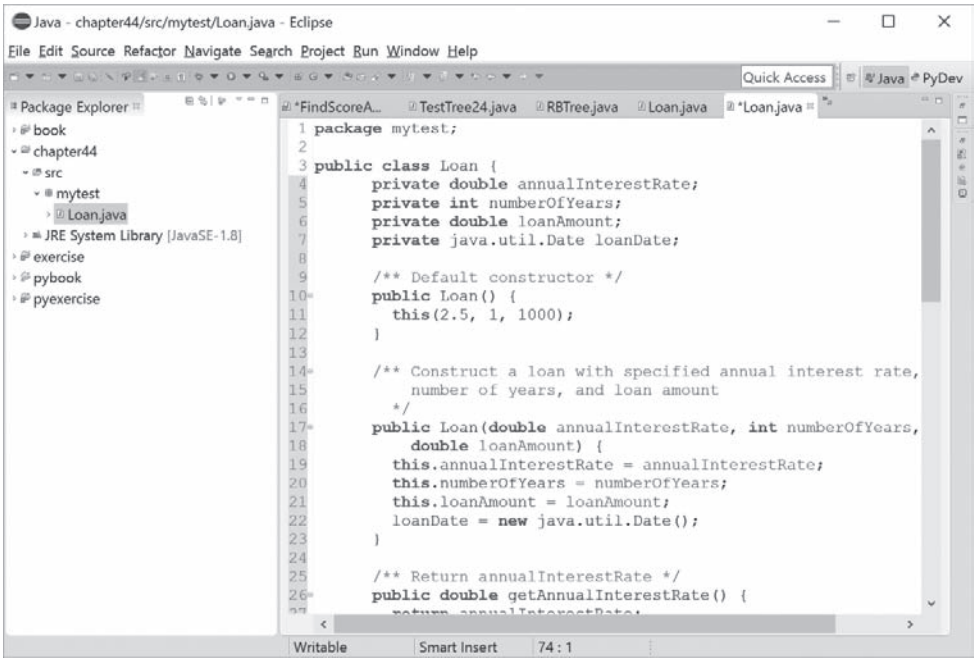
☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

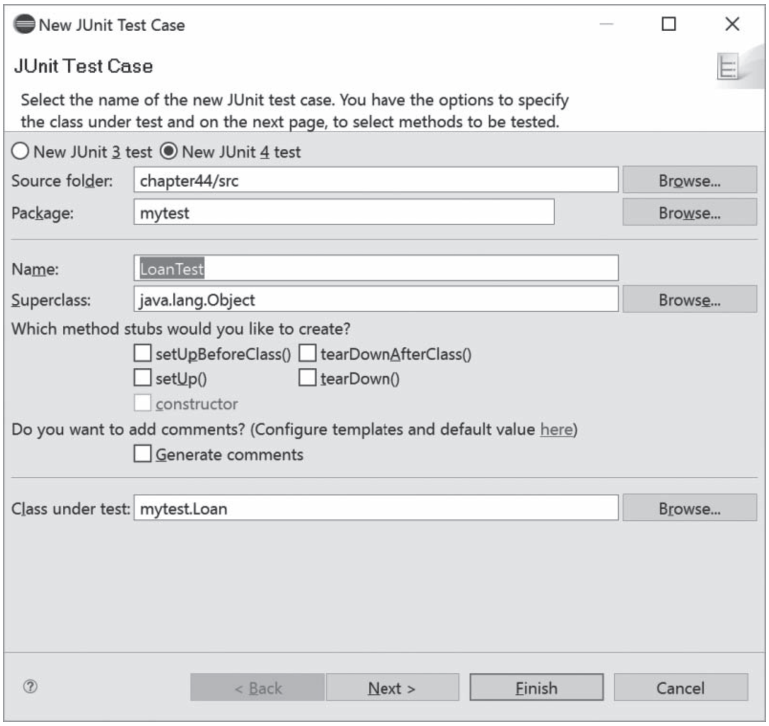
☐ Generate comments

FIGURE 44.13 The New Java Class dialog creates a new Java class.


Java - chapter44/src/mytest/Loan.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
PyDev
Package Explorer
book
chapter44
src
mytest
Loan.java
JRE System Library [JavaSE-1.8]
exercise
pybook
pyexercise
*FindScoreA...
TestTree24.java
RBTree.java
Loan.java
*Loan.java
Writable
Smart Insert
74 : 1

```
package mytest;

public class Loan {
      private double annualInterestRate;
      private int numberOfYears;
      private double loanAmount;
      private java.util.Date loanDate;

      /** Default constructor */
      public Loan() {
        this(2.5, 1, 1000);
      }

      /** Construct a loan with specified annual interest rate,
          number of years, and loan amount
        */
      public Loan(double annualInterestRate, int numberOfYears,
          double loanAmount) {
        this.annualInterestRate = annualInterestRate;
        this.numberOfYears = numberOfYears;
        this.loanAmount = loanAmount;
        loanDate = new java.util.Date();
      }

      /** Return annualInterestRate */
      public double getAnnualInterestRate() {
```

FIGURE 44.14 The Loan class is created.



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

○ New JUnit 3 test ● New JUnit 4 test

Source folder: chapter44/src

Package: mytest

Name: LoanTest

Superclass: java.lang.Object

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☐ setUp() ☐ tearDown()

☐ constructor

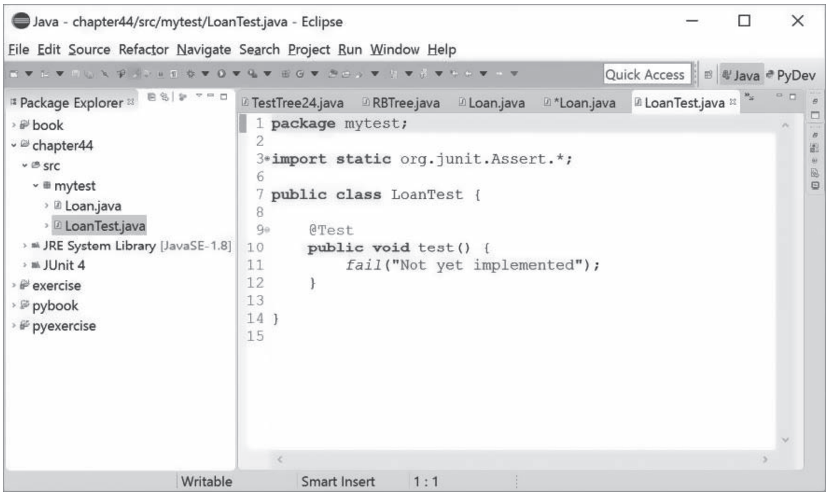
Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test: mytest.Loan

Browse...
Browse...
Browse...
Browse...
< Back
Next >
Finish
Cancel

Figure 44.15 The New JUnit Test Case dialog box creates a Test class.

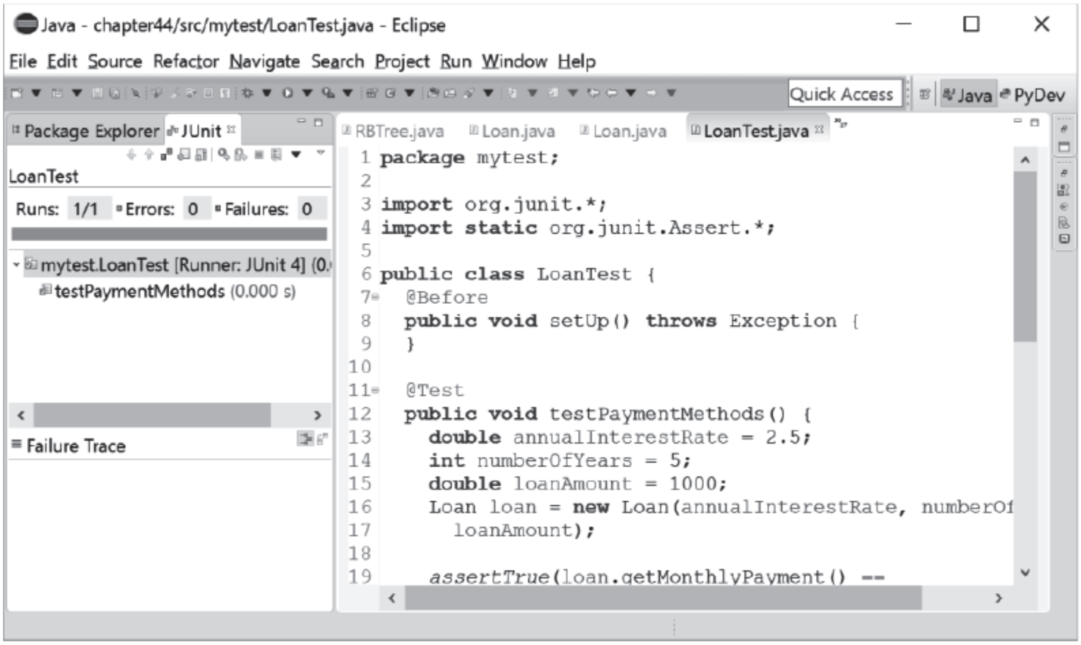


Java - chapter44/src/mytest/LoanTest.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
PyDev
Package Explorer
book
chapter44
src
mytest
Loan.java
LoanTest.java
JRE System Library [JavaSE-1.8]
JUnit 4
exercise
pybook
pyexercise
TestTree24.java
RBTree.java
Loan.java
*Loan.java
LoanTest.java
Writable
Smart Insert
1 : 1

```
package mytest;

import static org.junit.Assert.*;

public class LoanTest {

    @Test
    public void test() {
        fail("Not yet implemented");
    }

}
```

Figure 44.16 The LoanTest class is automatically generated.


Java - chapter44/src/mytest/LoanTest.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
PyDev
Package Explorer
JUnit
RBTree.java
Loan.java
Loan.java
LoanTest.java

LoanTest

Runs: 1/1 Errors: 0 Failures: 0

mytest.LoanTest [Runner: JUnit 4] (0.

testPaymentMethods (0.000 s)

Failure Trace

```java
package mytest;

import org.junit.*;
import static org.junit.Assert.*;

public class LoanTest {
  @Before
  public void setUp() throws Exception {
  }

  @Test
  public void testPaymentMethods() {
    double annualInterestRate = 2.5;
    int numberOfYears = 5;
    double loanAmount = 1000;
    Loan loan = new Loan(annualInterestRate, numberOf
      loanAmount);

    assertTrue(loan.getMonthlyPayment() ==
```

FIGURE 44.17 The test report is displayed after the LoanTest class is executed.

CHAPTER SUMMARY

1. JUnit is an open-source framework for testing Java programs.
2. To test a Java class, you create a test class for the class to be tested and use JUnit's test runner to execute the test class to generate a test report.
3. You can create and run a test class from the command window or use a tool such as NetBeans and Eclipse.

QUIZ

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

MyProgrammingLab™

44.1 Write a test class to test the methods `length`, `charAt`, `substring`, and `indexOf` in the `java.lang.String` class.

44.2 Write a test class to test the methods `add`, `remove`, `addAll`, `removeAll`, `size`, `isEmpty`, and `contains` in the `java.util.HashSet` class.

44.3 Write a test class to test the method `isPrime` in Listing 6.7 PrimeNumberMethod.java.

44.4 Write a test class to test the methods `getBMI` and `getStatus` in the BMI class in Listing 10.4.